EVALUATION OF BARNES' METHOD AND KRIGING FOR ESTIMATING THE LOW LEVEL WIND FIELD

THESIS

Michael W. Engel
1st Lt, USAF
AFIT/GM/ENP/99M-05

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GM/ENP/99M-05

EVALUATION OF BARNES' METHOD AND KRIGING FOR ESTIMATING THE LOW LEVEL WIND FIELD

THESIS

Michael W. Engel
1st Lt, USAF

AFIT/GM/ENP/99M-05

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

AFIT/GM/ENP/99M-05

EVALUATION OF BARNES' METHOD AND KRIGING FOR ESTIMATING THE LOW LEVEL WIND FIELD

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Meteorology

Michael W. Engel, B.S.

1st Lt, USAF

March 1999

Acknowledgments

I would like to take this time to acknowledge all of the support and understanding my loving family provided to me during the past 18 months. The hours were long and difficult, and I would never have succeeded without their help. I would also like to extend my appreciation to my thesis advisor, Lt Col Miner, and the rest of my committee members: Lt Col Walters, Prof. Reynolds, and Dr. Weeks. Lt Col Miner provided me with the direction and supervision necessary to succeed. Thank you, Prof. Reynolds, for your help in understanding Kriging and the statistical portions of my thesis. I could only learn so much from the books. Lt Col Walters, thank you for your help in directing the focus of the work. Your initial input was instrumental in getting the process started. Dr Weeks, I appreciate your time and insight. Your combined efforts enabled me to learn a great deal throughout the thesis process.

I would also like to take this opportunity to thank Jesus Christ for the strength and endurance required to complete such a challenging program.

Michael W. Engel

Table of Contents

Page

List of Figures

Figure **Page**

List of Tables

Table **Page**

EVALUATION OF BARNES' METHOD AND KRIGING

FOR ESTIMATING THE

LOW LEVEL WIND FIELD

Michael W. Engel, B.S.

First Lieutenant, USAF

Approved:

Cecilia A. Miner
Cecilia A. Miner, Lt Col, USAF
Chair, Advisory Committee

3 Mar 99
Date

Michael K. Walters, Lt Col, USAF
Member, Advisory Committee

3 Mar 99
Date

Daniel E. Reynolds
Member, Advisory Committee

3 Mar 99
Date

David E. Weeks, PhD
Member, Advisory Committee

3 Mar 99
Date

Abstract

This thesis evaluates two different methods of estimating a three dimensional wind field based upon a limited number of irregularly-spaced observations. This work was performed for the 45th Weather Squadron to determine how well the two methods worked and their potential for use in a visualization program. The two methods evaluated were Barnes' method and a method called Kriging, which is commonly used in geostatistics. Both of these estimation techniques were implemented and then evaluated to determine how accurate the estimates were that they created. The methods' accuracies were determined by withholding an observation from the observed wind field data set, performing the estimation, and then comparing the estimated value at the point of the withheld observation with the actual value withheld. These performance results were compared to determine which method produced a more accurate estimated wind field. Barnes' method proved to be the less complicated to implement, but Kriging provided a more accurate estimate. Both of the methods had a significant amount of estimation error associated with them. This large error casts serious doubt on their abilities to produce an accurate enough estimation to be useful in analyzing the low-level wind field.

EVALUATION OF BARNES' METHOD AND KRIGING FOR ESTIMATING THE LOW LEVEL WIND FIELD

I. Introduction

1. Motivation

a. 45th Weather Squadron Mission

The 45th Weather Squadron has the mission of *Exploiting The Weather To Assure Access To Air And Space*. To accomplish this mission they provide weather services for Cape Canaveral Air Station (CCAS), Kennedy Space Center (KSC), and Patrick AFB. These three facilities combine efforts to conduct an average of 60 attempted launches per year, of which 35 are successful launches, over 5,000 pre-launch operations, as well as NASA ferry flights and other military aviation operations. In addition to the billions of dollars of payload associated with these launches, the 45th WS is also responsible for weather warnings and watches that provide for the safety of 25,000 people and the resource protection of over 8 billion dollars worth of facilities (Roeder, 1998).

The 45th WS issues over 1200 lightning warnings/advisories per year as well as over 175 convective wind warnings. Weather warnings are special bulletins alerting customers to weather conditions that pose a hazard to life or property.

Weather advisories are notifications of weather conditions that could affect operations (Air Force Manual 15-125, 1997). These warnings and advisories are issued based upon a set of predetermined weather threshold values set by each customer of the 45th Weather Squadron. In addition to these standard warning and advisory thresholds, the 45th WS is also responsible for mission-specific forecasts covering launch operations with different requirements. Each launch operation has its own special set of launch criteria, including a variety of weather thresholds. The numbers of warnings and advisories provided above do not reflect these mission-specific forecasts.

Forecasting weather events that violate these standard thresholds results in the large numbers of weather warnings/advisories. The 45th WS's current forecasting capabilities have resulted in weather warnings and advisories having a 40 percent false alarm rate. This means that 40 percent of the time a forecasted weather event does not occur. Of those weather warnings and advisories that do verify, 10 percent fail to meet the desired lead time. Desired lead times are determined locally based on the amount of time a customer needs to take precautionary action, such as tying down aircraft or bringing workers to shelter. The high false alarm rate reflects the reality of having to err on the side of safety and caution when billions of dollars and thousands of lives are at risk. The bottom line is that 35 percent of all launches are either delayed or scrubbed as a result of weather (Roeder, 1998). The high percentage of false alarms and the high percentage of launches impacted by weather indicate the importance of weather forecasts to launch operations. They also indicate the huge negative

impact upon operations if a forecast is incorrect. This situation provides the motivation for finding better ways to produce accurate forecasts. This work provides the initial research for a proposed tool for improving the ability to forecast the local winds.

b. Local Wind Field

Forecasting wind conditions can be very difficult even in regions where the synoptic environment is relatively benign. This difficult task of forecasting the winds around Cape Canaveral is further complicated by the local landscape. Cape Canaveral has a very complex wind field created by a local topography composed of a mixture of land and water (Figure 1). Some of the larger bodies of water that dominate the landscape are the Atlantic Ocean, the Banana River, the Indian River, and the Mosquito Lagoon. This complicated land and water environment presents both a forecasting and an observational challenge. The lack of uniform terrain produces a complex wind field as the frictional properties of the local topography interact with the atmospheric pressure gradient. These complicated interactions increase the difficulty of forecasting the wind.

Not only is the local wind field difficult to forecast, it is also difficult to observe. The main observational difficulty results from the large number of wind sensors. These wind sensors are located over a vast area to provide the proper coverage for the launch pads and other resources. The wind data from these sensors currently is displayed in multiple locations and multiple formats, requiring weather personnel to mentally visualize how all the separate numbers combine

to create the wind field (Roeder, 1998). Ideally, all this data could be combined to produce one three-dimensional graphical display of the wind field. This display could then be used to augment both observational and forecasting capabilities, and to analyze the wind field to develop a better understanding of how the local conditions impact the wind.

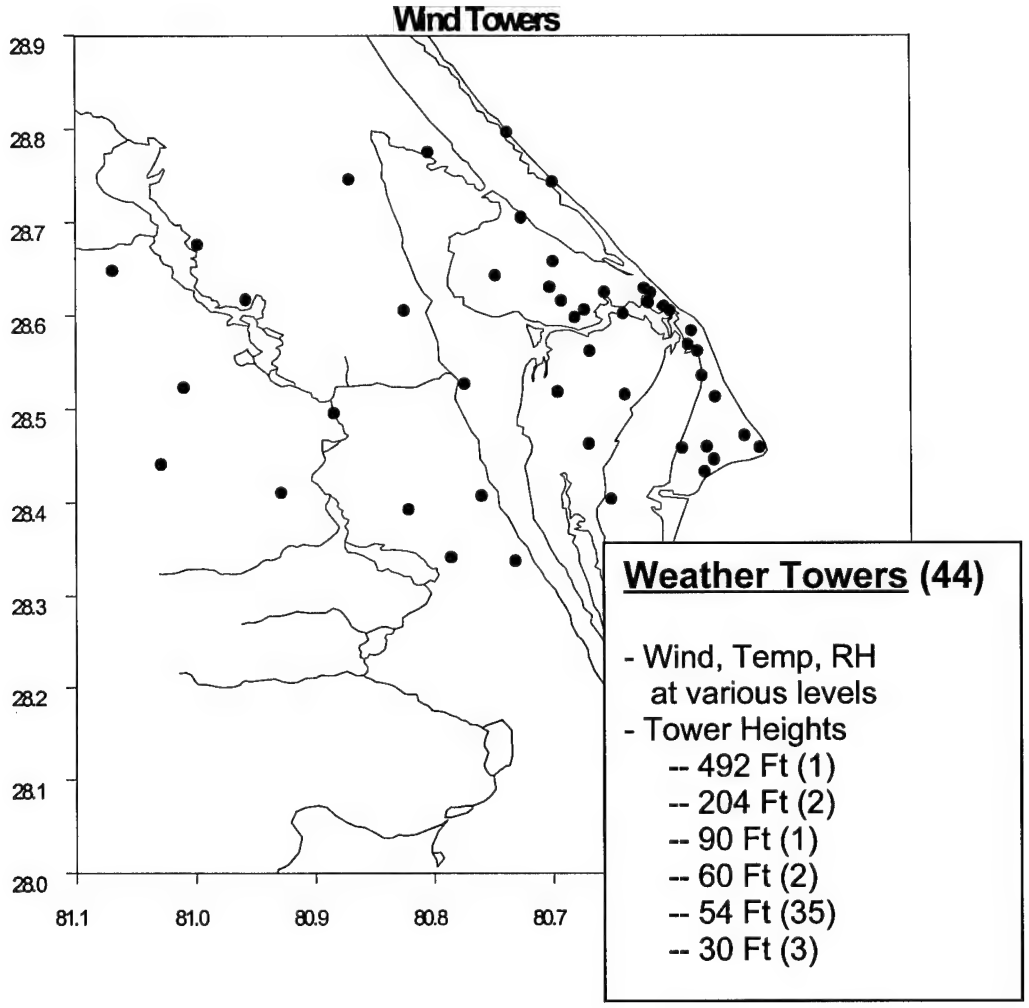

Figure 1 – Cape Canaveral Wind Tower Distribution

This figure provides a graphical depiction of the Cape Canaveral area and the spatial relationship between the wind towers

2. Problem Statement

To evaluate Barnes' method and Kriging to determine their accuracy in estimating the low-level wind field based upon the wind measurements from multiple wind sensors located on and around Cape Canaveral. Also, to evaluate their potential use as the estimation tool for a visualization program used to analyze the wind field in both time and space.

3. Scope

a. Thesis Scope

The 45th weather squadron desires a product that will produce a graphical display of the observed wind field at Cape Canaveral, which can be used both to visualize the current wind flow, and to analyze the wind field for forecasting purposes. To achieve this goal three major issues need to be addressed: data integration, estimation of the tree-dimensional wind field using the wind tower observations, and graphical representation of the data.

The scope of this work concentrates on the second of these three. The first portion of this project was not addressed because of hardware and budgetary constraints. To address the problems associated with having all the data transmitted to a central location would have required much of the work to be performed at Cape Canaveral, or for the system configuration there to be duplicated at AFIT. Neither of these was a realistic option due to lack of funds.

However, the procedures for archival of data took care of most of the issues associated with the problem of data integration. The format of the archived data is briefly discussed to provide a basic understanding of the data used to evaluate the estimation methods. The wind tower data archive files contained wind readouts of all the sensors, grouped by date and time. These problems of data integration resolved, although artificially, allowed for the main focus of the work to be centered upon the problem of estimating the wind field.

In meteorology this process of estimating a gridded field from non-spatially uniform observations is called objective analysis. This evaluation to determine the accuracy of the potential objective analysis methods must clearly be resolved before any issues about the display of the data can be addressed. Indeed, one could have the best display possible, but if the data is wrong, the display would not be useful.

This effort examines two different methods of estimating the three-dimensional wind field based upon the observations of Cape Canaveral's local wind tower network. The examination establishes the steps for performing the objective analysis and displaying the estimated wind field. It also provides a quantitative evaluation of which estimation method is the most accurate. The wind field estimate is computed for an area just large enough to contain all of the observational data provided by the wind tower data sets and is limited in the vertical to cover only the first five hundred feet due to lack of data above that level. There are a few wind profilers that can provide wind data above 1000 feet that can be included in future evaluations, but the scarce amount of data they

provide caused them to be of little help for evaluating which objective analysis method was better.

b. Constraints

Due to hardware limitations, financial restrictions, and time constraints, this work does not attempt to duplicate the actual sensor configuration in place at Cape Canaveral. The wind data is read in from data files and processed. Therefore, no attempt is made to produce an estimated wind field and display those results in "real-time." Instead the two objective analysis methods are evaluated to determine how accurate they are, which one works better, and whether or not they can be used in a graphical "real-time" system. Also, due to time constraints the final data set is imported into an existing graphics package, such as vis5d, for final display. Finally, this effort attempts to estimate the observed wind as accurately as possible, without modifying the observed values. Therefore, no efforts are made to ensure the observed wind field is in proper balance with any other atmospheric variables.

4. Summary

This chapter defines the importance of more accurate forecasts to the space program. With work on the new international space station underway, the ability to conduct launch operations reliably takes on an added significance. This work is an important first step towards the creation of a tool to display the observed three-dimensional wind field at Cape Canaveral. The remainder of this paper

provides the background material required to complete the work, how the evaluations were performed, the results of those evaluations, and the recommendations based upon those results.

II. Background/Literature Review

1. Chapter Overview

This chapter provides the background material for this project. Most of the background material and literature deal with the objective analysis methods being considered. Detailed mathematical derivations of the equations these methods are based upon is beyond the scope of this work. The appropriate references are included so those derivations can be examined if required. Both of these methods require the selection of parameter values to compute the estimated values. This chapter describes those parameters but does not address the steps used to determine the values of those parameters used for this application. Those steps and the reasons behind them will be presented in Chapter 3.

2. Objective Analysis Overview

To create a realistic three-dimensional visualization of the wind field, data from irregularly-spaced sensors must be analyzed and fit to a regularly-spaced grid. Figure 2 graphically depicts five irregularly-spaced observation points, labeled A through E, and a superimposed uniform grid. Finding an accurate method for using the data at points A -E, which are known, to estimate the values at each of the intersecting grid points, which are unknown, is not a trivial

undertaking. This task is commonly performed as one of the initial steps in numerical weather prediction and is called objective analysis. This name came about because in the past all weather maps were analyzed by hand. This was considered a subjective method because no two forecasters would produce the same analysis. The methods used to interpolate the meteorological variables across the uniform grid were developed to provide an "objective" analysis.

There has been much work concentrated on the task of developing objective analysis methods. The first efforts in meteorology were driven by the need to develop an automated process for initializing numerical weather prediction computer models (Daley, 1991:21). Panofsky created the first automated method based upon a scheme that used polynomial expansion to fit the observations. Gilchrist and Cressman made the next advancement by restricting the polynomial expansion's region of influence and suggesting the use of an initial or background field (Cressman, 1959). Barnes introduced his method in 1964 as an attempt to regain some of the signal that was being smoothed out of the data by the other methods (Barnes, 1964). Starting in the 1970s, one of the primary concerns became ensuring that the estimated fields produced by the objective analysis initialize numerical weather prediction computer models well. This initialization effort often involved modifying the data to ensure dynamic relationships between some of the atmospheric variables were satisfied (Daley, 1991:24). Because the desired output of the present study must be as close as possible to the observations, the emphasis is placed on methods that do not contain these additional features.

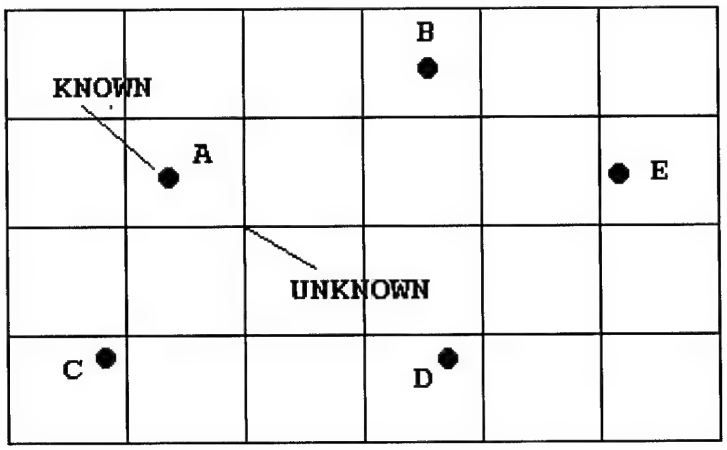

Figure 2 – Typical objective analysis grid
This figure graphically depicts the relationship between irregularly spaced observation points (A-E) and a superimposed regularly spaced grid.

One of the methods being evaluated was the method introduced by Barnes in 1964. Barnes' analysis scheme used the assumption that an atmospheric variable could be represented in two-dimensions as the summation of an infinite number of independent waves (Sen, 1997). In his own article Barnes indicated that his method was limited to applications where the data sampling was "reasonably uniform" (Barnes, 1964). The distribution of the data points shown in Figure 1 indicates that this scheme might not be well suited for this project.

In addition to Barnes' method the 45th WS wanted Kriging or Bratseth to be evaluated to determine if they could produce a more accurate estimated wind field. The first method, Kriging, is well established in the geosciences and has

recently been considered as a valid method for solving atmospheric problems (Sen, 1997). Kriging makes use of a semi-variogram to allow weight functions to be based on the variation of the observed variable as a function of distance (Clark, 1979). The second method, the Bratseth method, derives the weights by using the correlation function for the forecast error (Sen, 1997).

This project compared Barnes' method and Kriging to determine which method produced the more accurate estimated three-dimensional wind field. Time did not permit the examination of both Bratseth and Kriging. Kriging was chosen over Bratseth because most current numerical weather prediction models use optimal interpolation, or statistical interpolation, techniques that are similar to Kriging (Daley, 1991: 99). Contact with personnel in the MM5 model section at the Air Force Weather Agency, Offutt AFB, NE revealed that the preferred method of objective analysis continues to be the use of optimal interpolation schemes that take into account the observation error covariances (Williams, 1998). The remainder of this chapter will provide the background material required for understanding the Barnes and Kriging methodologies.

3. Barnes' Analysis

a. Overview

Barnes proposed his method of estimation in 1964, in an effort to improve the details represented in the interpolated field. The Barnes objective analysis belongs to the class of successive approximation, or successive correction,

methods. This method is widely used because it is versatile, fast, and simple (Mullen, 1993). Another benefit is that it can be used in cases where there is no reasonable background field available, which is the case for Cape Canaveral (Daley, 1991:90). Barnes' method is based upon the assumption that a two-dimensional distribution of an atmospheric variable can be represented with a summation of an infinite number of independent harmonic waves, known as a Fourier integral (Barnes, 1964). Barnes states that this method is dependent upon the data density and therefore is best used in areas where the data density is fairly uniform. Even though Barnes emphasizes the importance of the uniform data dependency, he does indicate the possibility of being able to partially compensate for non-uniformity.

The density of the observations, as depicted in Figure 1, is not very uniform. The Cape Canaveral wind tower sensors are configured such that there are areas with greater density than others. Nevertheless, Barnes' method is used in a wide variety of situations in meteorology, such as synoptic objective analyses, which also do not have a very uniform data distribution. In spite of its limitations, one desirable feature of the Barnes' method is that his algorithm converges to the observations at the observation locations (Daley, 1991:92). This attribute meets the requirement of having the estimated wind field match the observed wind field as closely as possible.

b. Mathematical Formulation

Barnes begin the development of his method with the representation of the atmospheric quantity, $f(x, y)$, as the following smoothed function:

$$g(x, y) = \int_0^{2\pi} \int_0^{\infty} f(x + r\cos\theta, y + r\sin\theta)\omega\, r\, dr\, d\theta,$$

where the smoothing is a result of the weight factor,

$$\omega = \left(\frac{1}{4\pi k}\right)\exp\left(-\frac{r^2}{4k}\right).$$

In this equation θ and *r* are polar coordinate variables with origin centered at *(x,y)*, *r* is the distance between the observation and the point where the estimation is being computed, and *k* is a smoothing parameter related to the data density (Barnes, 1964). This function uses $g(x, y)$ to represent $f(x, y)$ by integrating the values of $f(x, y)$ at all the surrounding locations multiplied by the weight factor. To obtain a maximum weight being assigned when the value of *r* = 0, Barnes rearranged the equations as follows:

$$g(x, y) = \int_0^{2\pi} \int_0^{\infty} f(x + r\cos\theta, y + r\sin\theta) \times \left(\frac{\eta}{2\pi}\right) d\left(\frac{r^2}{4k}\right) d\theta,$$

where

$$\eta = \exp\left(-\frac{r^2}{4k}\right).$$

This weight function, η, is a function of the distance between the observation and the grid point where the estimation is being calculated. η decreases exponentially as r increases regardless of the direction. At r = 0, η is maximized at 1. η approaches 0 as r approaches ∞, meaning that every observation regardless of distance provides some degree of influence upon the value of the variable at the estimation grid point. It is possible to define a radius of influence, beyond which all weights are assigned the value of zero. These equations are not easy to implement because the analytical form of $f(x,y)$ is unknown, and the function cannot be integrated to infinity. These reasons prompted Barnes to use the following approximation:

$$g(x,y)=\frac{\sum_{j}^{M}\eta(r_j)\cdot f_j}{\sum_{j}^{M}\eta(r_j)},$$

where M represents the number of data points available within the region of influence, f_j is the value of the observation, and $\eta(r_j)$ is the weight function. This practical equation results in taking the weighted average of the *M* observations.

The next item to consider is how to determine the 4*k* used in the η term, which depends upon the minimum separation between observations as well as how much smoothing is desired (Mullen, 1993). The average distance between observation locations can be calculated as follows:

$$\bar{d}=\left(\frac{Area\ Of\ Domain}{Number\ Of\ Observations}\right)^{\frac{1}{2}}.$$

This equation provides a mechanism to attempt to compensate for non-uniform data density, and the $\bar{d}$ value establishes the upper and lower limits for $\sqrt{4k}$. If $\sqrt{4k} < \bar{d}$ then the analysis will be too noisy, and if it's greater than $\bar{d}$, the analysis will be too smooth. One empirically-derived relationship shows $\sqrt{4k}$ should be 1.33 times $\bar{d}$ (Mullen, 1993).

c. Implementation of Barnes' Method

The algorithm implemented was as follows (Mullen, 1993):

1. Start with the observations and the distances from the grid point being estimated at and the observation points.

$$F_k = \text{the value of the observation at point i}$$
$$\mathrm{R}_{\mathrm{i,j,k}} = \text{distance from grid point (i, j) to observation point k}$$

2. Calculate the first pass weights for each grid point. During this process a new and different set of weights is calculated for the observations for each grid point. It takes two steps to calculate these weights. The first pass determines a raw weight by taking the exponentiation of the negative of the distance squared divided by the 4k value.

$$W_{i,j,k} = \exp\left(\frac{-R^2{}_{i,j,k}}{4K}\right)$$

The second part of this process is to normalize these raw weights so that they sum to one. This new set of weights is obtained by dividing each raw weight by the sum of the raw weights.

$$B_{i,j,k} = \frac{W_{i,j,k}}{\sum_k W_{i,j,k}}$$

3. At this point the first pass estimate can be calculated for each grid point by summing the values obtained by multiplying the observed value with the weight assigned to that observation.

$$G_{i,j} = \sum_k B_{i,j,k} F_k$$

Note: Steps 2 and 3 are performed interactively – the weights are calculated for a given grid point, the estimate is made, and then the next grid point is worked on.

4. After the first estimate has been calculated, a correction factor is computed at each observation point. The correction factor is found by taking the average value of the four surrounding grid points and subtracting it from the observed value at that point.

$$FC_k = F_k - \overline{G}_k$$

5. Next the second pass weights are determined using the same method as in step 2. The algorithm implemented in this thesis used a new value of 4K that was one third the original value used in the first pass calculations. This value helps the technique converge faster (Daley, 1991:92).

$$W'_{i,j,k} = \exp\left(\frac{-R^2_{i,j,k}}{4K'}\right)$$

$$B'_{i,j,k} = \frac{W'_{i,j,k}}{\sum_k W'_{i,j,k}}$$

6. The second pass estimate is now calculated on the correction factors computed in step 4. This estimate is arrived at using the same method as in step 3.

$$GC_{i,j} = \sum_{k} B'_{i,j,k} FC_k$$

7. The final estimate at each grid point is then computed by adding the first estimate at that grid point to the second estimate at the same grid point.

$$GF_{i,j} = G_{i,j} + GC_{i,j}$$

Steps 4 – 7 can be repeated until the correction factors fall below some predetermined error threshold. The algorithm implemented in this thesis only performs them once because the benefits of successive iterations diminish rapidly with each iteration (Mullen, 1993).

4. Kriging

a. Overview

Kriging is similar to the Barnes' method because it also assigns a set of weights to each observation in an attempt to describe the influence of each observation on the value at a location some distance from it. The major difference between the two methods is that Kriging attempts to use a description (semi-variogram) of how the observed data varies as a function of distance, and then attempts to calculate the Best Linear Unbiased Estimator (BLUE). The BLUE is obtained by minimizing the estimation error variance (Clark, 1987:106).

b. Semi-variogram/Covariance function

The first task to be performed in the Kriging process is to describe the variability of the data as a function of distance. The tool used to help define this variability is called a semi-variogram. A semi-variogram is one half of the variogram, which is the variance of the differences between the values at two observation points (Clark, 1987:5). It is calculated using

$$\gamma(h)=\frac{1}{2n}\sum[g(x)-g(x+h)]^2 \quad ,$$

where $g(x)$ is the observed value at location *x*, and $g(x+h)$ is the observed value at a distance of *h* from location *x*.

These values are calculated for each set of distances between observations. An example from the mining field will make this method more clear. In order to take mineral samples, boreholes are drilled at specified distances. If the holes are set up in a grid so that the distance between holes, *h,* is 50 feet, one is able to define a set of sample pairs that were taken 50 feet apart from each other. This set of data pairs allows the semi-variogram value to be calculated for that distance value. By skipping holes, one can then establish a set of values based on holes 100 feet apart. This process of defining sets based on the distance between the observations is only limited in a practical nature. That is, money and time prevent taking all the observations that might be desired. Once a sufficient number of sets have been analyzed, the semi-variogram data can be plotted on a scatter plot with the semi-variogram data on the vertical axis and distance on the horizontal (Figure 3). This scatter plot provides a visualization of

the variability of the data as a function of distance between two observations. Figure 3 is an example semi-variogram showing how the u and v components of the wind varied as a function of distance. In this example the fractional distances are a result of the distance between the observation locations being calculated in degrees of latitude squared by using the following equation:

$$d = (\Delta\phi)^2 + \cos^2 \bar{\phi}(\Delta\lambda)^2$$

where ϕ is the latitude and λ is the longitude. The use of this equation to calculate the distance is based upon the assumption that Pythagorean's theorem can be applied (See Appendix C). This approximation does not introduce a significant error in the distances being used. The use of the radius of the earth to calculate "more accurate" distances in meters also introduces uncertainties because of the estimation involved with measuring the earth's radius. The impact these small errors has is minimal because the relative differences between distances are the key element used to establish the weights (Mullen, 1993).

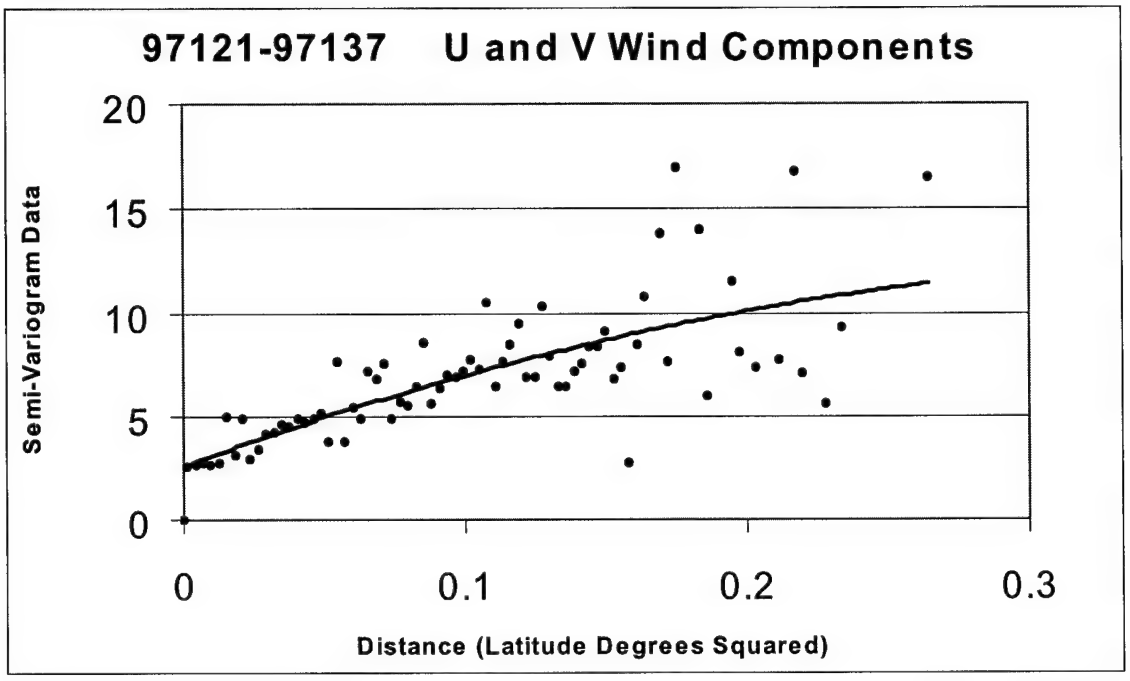

Figure 3 – Semi-variogram
This figure shows the scattergram produced when semi-variogram values are plotted verse distance.

Plotted on the scatter plot with the semi-variogram data is the best fit 2nd degree polynomial. Subjectively, it can be seen that the curve fits the data fairly well up to the 0.15 distance point; then the fit becomes less clear. In the ideal case the polynomial will intersect the origin. Theoretically, there should be no difference between two observations taken at some infinitely small distance from each other. Unfortunately, real world observations often conflict with the theoretical due to observational errors; therefore, the mathematical models that will be used to fit the semi-variogram data allow for a step discontinuity at the origin through a parameter called a "nugget." This nugget is essentially the y

intercept of the model fit to the data. As the distance increases the line reaches a point called a "sill." This is the maximum semi-variogram value and is the value given for large distances. Once the sill has been reached the semi-variogram data is considered constant. The point where the sill is reached is called the "range." In Figure 3 the nugget value would be approximately 2.5. The sill and the range are not displayed on this graph because there was no data for the distances the model suggests would be the range. One can visualize the curved line maximizing at around 14; this would be the sill. The distance value (approximately 0.45) where the sill is realized would be the range (Isaaks and Srivastava, 1989:292).

Once the semi-variogram data has been plotted, a mathematical model can be fit to it. A model is necessary because values will be needed for distances other than those measured. Selecting a model to fit the data requires some restrictions be adhered to. Kriging, as will be shown in the next two sections, involves the solution of n+1 equations, with n+1 unknowns (where n is the number of observations). It is desirable for these equations to have only one stable solution (Isaaks and Srivastava, 1989:371). One way this can be guaranteed is to use functions that are known to create a positive definite matrix (Isaaks and Srivastava, 1989:372). The use of positive definite matrix ensures a real solution when inverting the matrix used to perform the Kriging weight calculations (Kreyszig, 1993). Therefore, the Gauss-Jordan method which provides a straightforward method for inverting the resultant matrix, would not fail to work properly (Chapra and Canale, 1988:250). Four models that meet this

requirement are the spherical, the exponential, the Gaussian and the linear model. In the event that the parameters of any one of these models cannot be adjusted to create a satisfactory fit, any linear combination of the models will still satisfy the condition of being positive definite (Isaaks and Srivastava, 1989:375-376).

Another consideration when fitting a model to the semi-variogram data is whether or not the variation in the observations is a function of direction as well as distance. This study does not consider the case where the model considers the variation as a function of direction. Instead it considers only the isotropic case in which the sample semi-variogram is viewed as omni-directional, or the same in all directions. The use of the isotropic model is dictated by the layout of the wind tower network. The network layout prevents being able to isolate what portion of the variable's change is due to its displacement in either latitude or longitude. The variations are measured in an omnidirectional sense making the omnidirectional model the only choice. Isaaks and Srivastava consider the isotropic case easier to model because it is "better behaved" (Isaaks and Srivastava, 1989:375).

Clark (1987) and Isaaks and Srivastava (1989) provide the standard forms for all four of the models. Only the exponential equation is provided here to aid in understanding how Kriging is implemented. The exponential model has the following form:

$$\gamma(h) = 1 - \exp\left(-\frac{3h}{a}\right),$$

where *a* is the distance at which the semi-variogram value reaches 95% of its sill value, and *h* is the distance between the two points of concern.

With a model fit to the semi-variogram, the last remaining task to be accomplished is to determine the corresponding covariance function. Isaaks and Srivastava provide a standard form of both the exponential semi-variogram model and its corresponding covariance function with all the parameters required to account for the "nugget effect," range, and sill. The complete form of the exponential semi-variogram model is:

$$\hat{\gamma}(h)=\begin{cases}0 & \text{if } |h|=0\\ C_0+C_1\left(1-\exp\left(-\dfrac{3h}{a}\right)\right) & \text{if } |h|>0\end{cases}.$$

The corresponding covariance model is:

$$\tilde{C}(h)=\begin{cases}C_0+C_1 & \text{if } |h|=0\\ C_1\exp\left(-\dfrac{3h}{a}\right) & \text{if } |h|>0\end{cases},$$

where C_0 is the nugget effect, $C_0 + C_1$ is the sill, and *a* is the range. See Figure 4 for an example that shows the relationship between the covariance model and the semi-variogram model.

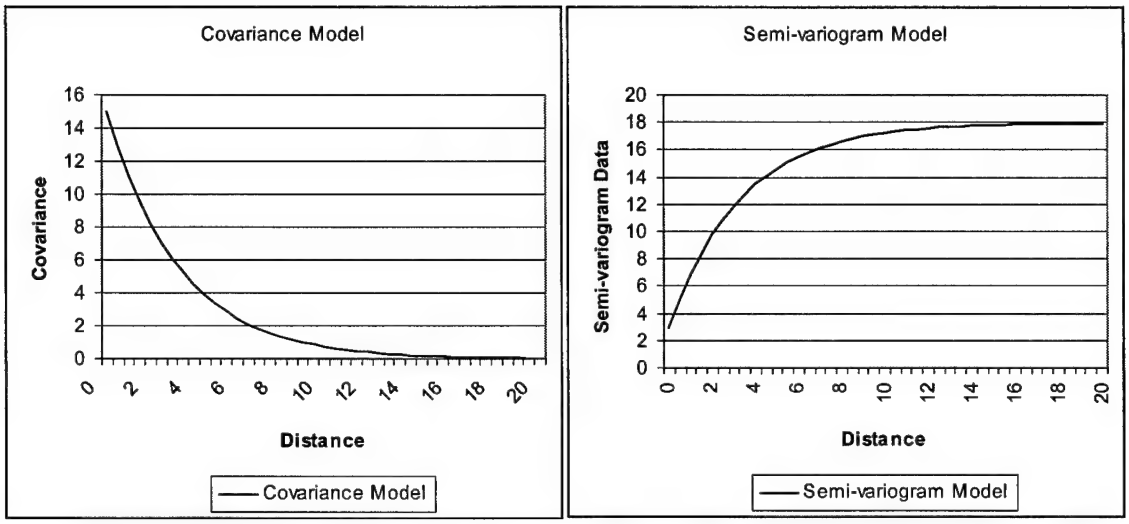

Figure 4 – Example Semi-variogram/Covariance Model Plots
This figure shows the relationship between the covariance model and the semi-variogram model. Note that the covariance model approaches zero as distance increases.

c. Mathematical Formulation

This section examines the error variance and how it is minimized to produce the Kriging method. The error variance, σ^2, of a set of estimates with a mean error of 0 can be written as

$$\sigma^2 = \frac{1}{k}\sum_{i=1}^{k}\left[\hat{v}_i - v_i\right]^2$$

where v_i are the actual values and $\hat{v}_i$ are the estimated values. This equation is not very useful because the actual values are not known. Therefore, an equation similar to the one used in the development of the Barnes method must be employed:

$$\hat{V}(x_0) = \sum_{i=1}^{n} \omega_i V(x_i).$$

This equation sets the value of the estimate, $\hat{V}(x_0)$, to be equal to the sum of the observations, $V(x_i)$, multiplied by a weight, ω_i. Subtracting this estimated value from the true value yields the error associated with the estimate. Isaaks and Srivastava (1989:284) show how the variance of this error can be expressed as

$$\overline{\sigma}^2{}_R = \tilde{\sigma}^2 + \sum_{i=1}^{n}\sum_{j=1}^{n} \omega_i \omega_j \tilde{C}_{ij} - 2\sum_{i=1}^{n} \omega_i \tilde{C}_{i0} ,$$

where $\tilde{\sigma}^2$ is the estimated variance, $\tilde{C}_{ij}$ is the model covariance values calculated with the distance between observation points, and $\tilde{C}_{i0}$ is the model covariance values calculated based upon the distance between the observation and the estimation location. This equation provides an expression for the error variance as a function of *n* unknowns, the *n* weights. The minimization of this error variance can now be accomplished by setting the first partial derivatives of these n equations equal to 0. This current set of equations does not restrict the sum of the weights to be equal to 1. Therefore another term, called a Lagrange Parameter must be added, yielding:

$$\overline{\sigma}^2{}_R = \tilde{\sigma}^2 + \sum_{i=1}^{n}\sum_{j=1}^{n} \omega_i \omega_j \tilde{C}_{ij} - 2\sum_{i=1}^{n} \omega_i \tilde{C}_{i0} + 2\mu\left(\sum_{i=1}^{n} \omega_i - 1\right).$$

The last term is equal to zero because of the requirement for the summation of the weights to be equal to 1. The new variable, μ, is calculated as the n+1 term in the weight matrix, and is only used to calculate the actual error variance. Minimizing these equations yields the final set of n+1 equations.

$$\sum_{j=1}^{n} \omega_j \tilde{C}_{ij} + \mu = \tilde{C}_{i0} \quad \forall \ \mathrm{i} = 1,\ldots,\mathrm{n}$$

$$\sum_{i=1}^{n} \omega_i = 1$$

where C_{ij} is the model covariance value between observation i and observation j, and C_{i0} is the model covariance value between observation i and the grid point where the estimation is being performed. These equations can be expressed in the following matrix form, which is considered ordinary Kriging:

$$C \bullet w = D,$$

where the C matrix contains the covariance values based upon the distance between the observation points, the D matrix contains the covariance values based upon the distance between the observation point and the estimation point, and when solved the w matrix contains the desired set of weights associated with each observation. An expanded form of the matrix would have the following form:

$$\begin{bmatrix} \widetilde{C}_{11} & \cdots & \widetilde{C}_{1n} & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \widetilde{C}_{n1} & \cdots & \widetilde{C}_{nn} & 1 \\ 1 & \cdots & 1 & 0 \end{bmatrix} \bullet \begin{bmatrix} w_1 \\ \vdots \\ w_n \\ \mu \end{bmatrix} = \begin{bmatrix} \widetilde{C}_{10} \\ \vdots \\ \widetilde{C}_{n0} \\ 1 \end{bmatrix}.$$

d. Implementation of Kriging

This method is implemented using the following steps.

1. Use the observations to determine which of the four standard models, or linear combination of them, provides the best covariance model to describe the variability of the data. Once the model has been selected determine the appropriate values to use for its parameters.
2. Use the covariance model to complete the C and D matrix.
3. Invert the C matrix.
4. Multiply the C and D matrix together to get the weight matrix.

5. Get the final estimate by summing the observations multiplied by their associated weights.

6. Repeat all the steps for each new set of observations. When moving to a new estimation point and still using the same observations, the D matrix must be build for each estimation point; then repeat steps 4 and 5.

5. Summary

This chapter has provided the background material required to follow the implementation of the objective analysis methods. The Barnes method was shown to have weights assigned to observations that decrease exponentially as the distance increases. Kriging was shown to be a method that attempts to take into account the natural variation of the data as a function of distance. Kriging is also supposed to provide the best linear unbiased estimator because of the minimization of the estimated error variance in its derivation (Clark, 1987: 106). Actual implementation issues will be discussed in the following chapters.

III. Methodology

1. Chapter Overview

This chapter provides a detailed description of the thesis work. A brief description of the program design is provided, but the emphasis is placed on how the methods described in Chapter 2 were implemented. As stated above, the primary goal of this project is to evaluate the two different objective analysis methods to determine how accurate they are at estimating the low-level wind field and which will provide the more accurate estimation of the wind field. The major tasks that needed to be accomplished were ingesting the archived data, performing the horizontal objective analysis, and estimating the error.

Before continuing, it's important to know that the objective analysis methods were applied only on the horizontal plane at the height the observations were taken. Once this two-dimensional analysis was performed, a different interpolation method was used to fit the data to levels that had equal vertical spacing. The method used for this vertical estimation was cubic-splines. There are a couple of reasons why the objective analysis methods were applied only to the horizontal plane. One of the main reasons is that atmospheric variables normally vary much less in the horizontal than in the vertical. Also, atmospheric conditions, like a temperature inversion, can exist that will produce a barrier between winds aloft and winds at the surface. This condition will prevent the

winds at one level from exerting any significant influence over the winds at another (Garratt, 1992:3). Additionally, vertical interpolation has its own unique set of problems associated with it that were beyond the scope of this thesis.

2. Program Design

Before proceeding with these tasks, the computer program design must be considered. The solutions to the algorithms were implemented using the C++ programming language. The use of this language allowed for the encapsulation of data and the functionality required to manipulate it into program units called objects. Lower level objects were created to handle individual wind observations, sensor data, and many other program elements. These lower level objects were then combined to create objects for the entire observation set, as well as the Barnes' and Kriging algorithms. The general program flow is

1. Read in the sensor data.
2. Read in the entire observation file (all time steps).
3. Perform the objective analysis, one time step at a time.
4. Output those results to a temporary data file.
5. Read in the temporary data file and perform the vertical interpolation, once the objective analysis is completed at all the observation levels.
6. Print the final 3-D wind field to the output file.
7. Convert the data to a vis5d file and start vis5d, to visualize the data.

These routines are contained in the files *obanal.h* and *obanal.cc*. The file *obanal.h* contains the entire object descriptions, while *obanal.cc* contains all of the executable code (Appendix A). The code was written to follow ANSII standard to ensure its portability. The primary development was on a Pentium-based PC using Borlands Turbo C++. The code has been ported to the SUN workstations located in the AFIT Weather Laboratory, where it has compiled and executed.

3. Data Integration

Next, implementation issues are addressed. Data integration proved to be a straightforward process due to the use of archived data. The ASCII data files provided by the 45th WS contained the wind observations already sorted by time. Each observation included the date, time, tower identification number, and height of the observation. Using the station identification and a lookup table the latitude and longitude of each wind tower were obtainable. This portion of the project involved reading the raw ASCII data set for the wind tower sensor system. The data files came in self-extracting zip files, each containing 10 files. Each of these files contained 1 day's worth of observations. The observations were taken at 5-minute intervals, yielding 288 time slots in a 24-hour period. The only processing performed upon the data was to exclude observations that fell outside of 3 standard deviations from the mean value.

Data archival procedures ensured that all the observations were already grouped by time eliminating the need to make sure observations fell within a

predetermined time window. No other processing of the data needed to be performed to make sure the data balanced in the appropriate meteorological equations because the data was not being used to initialize a computer model. Another reason further processing was not performed was because of the desire for the estimated wind field to remain as close as possible to the observed wind.

The wind observations were recorded in the standard meteorological format of wind direction and wind speed in knots. This data was converted to the vector components. The *v* component is the south-north component of the wind, and the *u* component is the west-east component of the wind. The actual objective analysis methods were used to compute the estimated *u* and *v* components of the wind based upon the observed *u* and *v* components of the wind.

4. Objective Analysis Implementation

a. Overview of Implementation

This section details the steps taken to implement both Barnes' method and Kriging. It provides all of the reasoning behind the selection of the parameters used to reach the final solution. One underlying assumption for both of these methods all is that error associated with the estimated wind value is unbiased. This unbiased condition means that the average error associated with the estimation would equal zero.

b. Barnes

The Barnes method did not require a great deal of parameter selection. The algorithm described in Chapter 2 was implemented without any significant variations. The one parameter that needed to be determined was what 4k value to use. The method implemented attempts to compensate for the non-uniformity of the data spacing by determining the average distance between stations using the formula provided in Chapter 2. The 4k value had to be determined dynamically because the number of observations recorded at each level was different. The dynamic calculation also compensated for observations that might not be available due to a variety of reasons.

c. Kriging

The next method implemented was Kriging. The Kriging method is based upon having a good understanding of how the observed variable varies as a function of distance. Finding this relationship was the first issue addressed.

To find this relationship one time step of observations was read in and the semi-variogram data was calculated. To plot this data, the data was collected in bins based upon distance. The use of bins was necessary because of the distribution of observation points not being uniform. All the semi-variogram values calculated that fell within the same specified range of distance values were place into the same bin and the average value of the difference squared between the two observations was computed. The initial selection of the bin size

was based upon efforts to balance smoothing of the data and having sufficient data to produce the semi-variogram graph. This initial bin size was calculated by taking the range of distances and dividing it by 100. The smallest distance was 0.000147 degrees squared and the largest distance between observation stations was 0.264547 degrees squared. This number was rounded to 0.28 degrees squared, yielding an initial bin size guess of 0.0028 degrees squared. Figure 5 shows the number of observational pairs whose separation falls within each distance bin. The histogram function within Microsoft Excel suggested a bin size of 0.0115 degrees squared as an optimum bin size to minimize the number of empty bins and have as equal a number in all the remaining bins as possible. This value of bin size was almost 4 times as large as the one initially selected. This suggested bin size resulted in 152 of the 555 distances being placed in the second bin. Using this size bin would cause over one fourth of the data to be averaged into one value greatly smoothing the data. The 0.0028 bin size had a maximum of 48 in one bin.

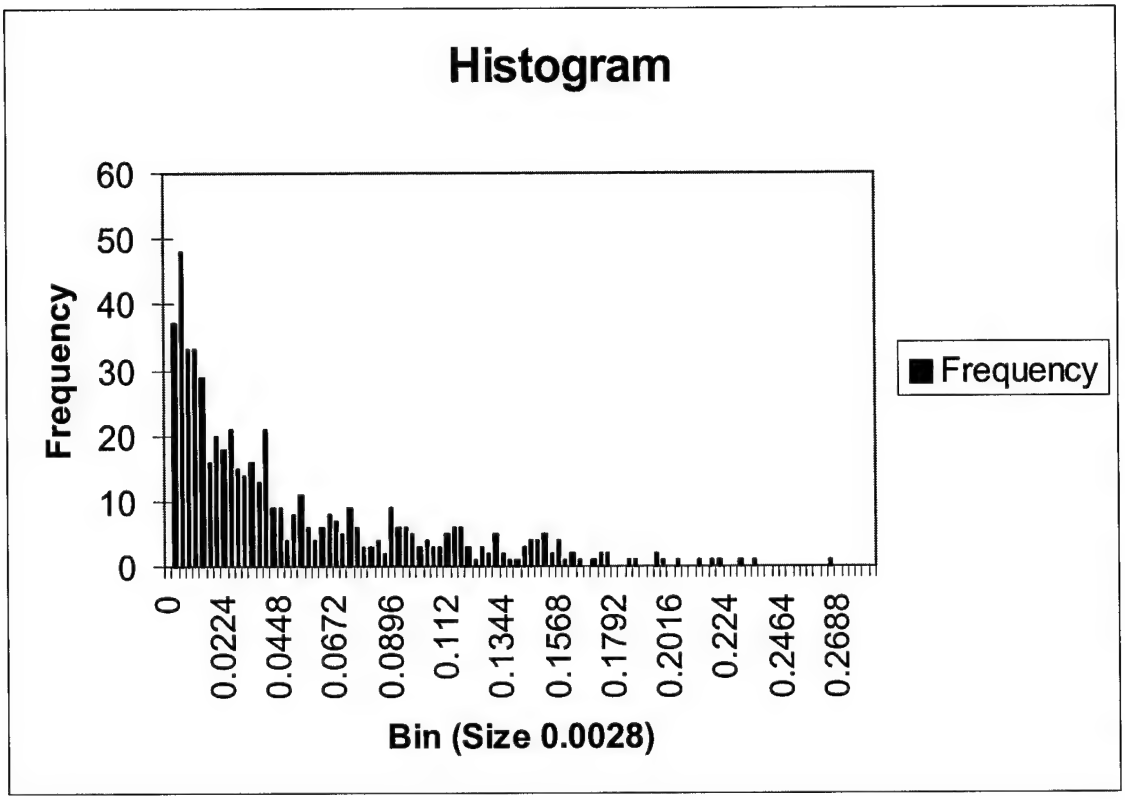

Figure 5 - Distance Histogram
This histogram shows the number of data pairs that fell into each bin size. The semi-variogram values associated with these data pairs are averaged to produce one value per bin. The distance is in units of degrees squared.

In an effort to further evaluate the bin size, different bin sizes were used to see how they affected the ability of a function to be fit to the scatter gram of the semi-variogram data. The measure used to determine which bin size was best was the R^2 value on the second order polynomial trendline added to the data. The first size looked at was half the initial size. As expected, halving the bin size increased the variability of the data and reduced the fit; the R^2 got smaller. Next, a bin size twice as large was selected. The expected result of this was that the data would be smoothed more and the fit would improve. The R^2, however, got

smaller indicating that the trendline function had a more difficult time fitting a function to this data. As a result, the original bucket size of 0.0028 was used throughout the process.

This initial data for one time slice was then plotted on the scatter gram shown in Figure 6. To use this graph as a basis for selecting a model, a function needed to be fit to the plotted data. Examination of the data revealed that no function could reasonably be fit to it. In the mining world the semi-variogram data was calculated by averaging several measurement differences for the same distance. The Cape Canaveral situation prevented a sufficient number of measurements from being taken at the same distance apart. Isaaks and Srivastava (1989) used a couple of examples of data that was averaged in time to produce the semi-variogram data. So, this concept was applied to the Cape Canaveral problem in combination with the distance averaging to produce semi-variogram data derived from both distance and time averaging. Figure 7 shows the first scatter plot of this data combining 9 days of data. This averaging did produce a scatter gram containing data points that a function could be fit to.

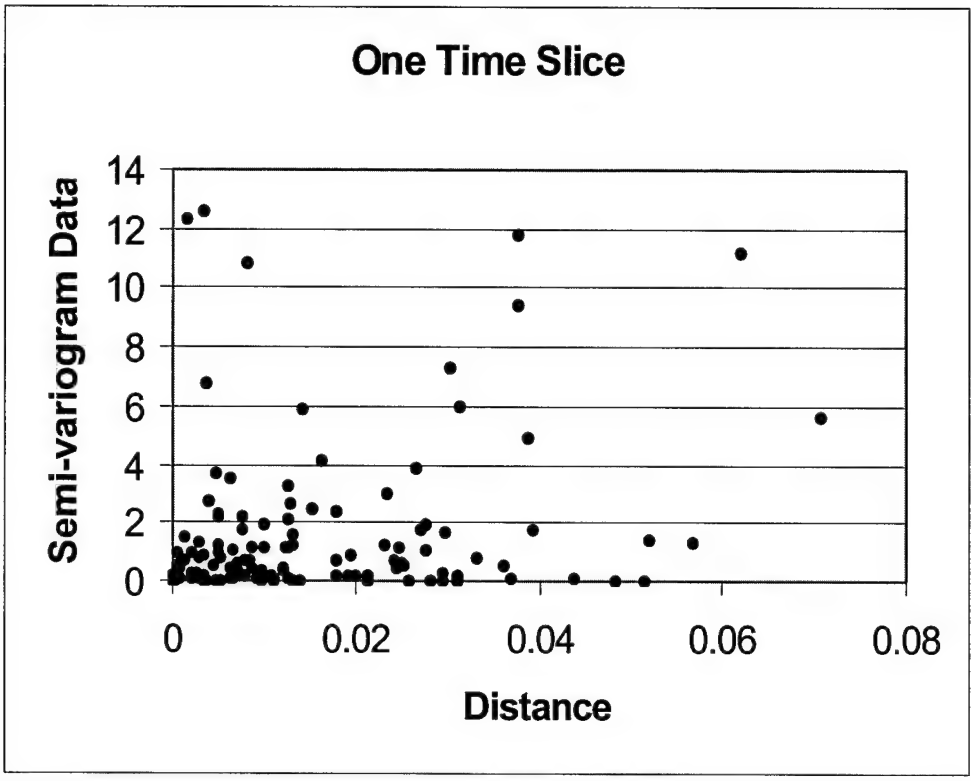

Figure 6 - Semi-variogram (1Time Slice)
The semi-variogram data plotted on this graph is for one time slice of observations. The distance is in units of degrees squared and the semi-variogram data in units of knots squared.

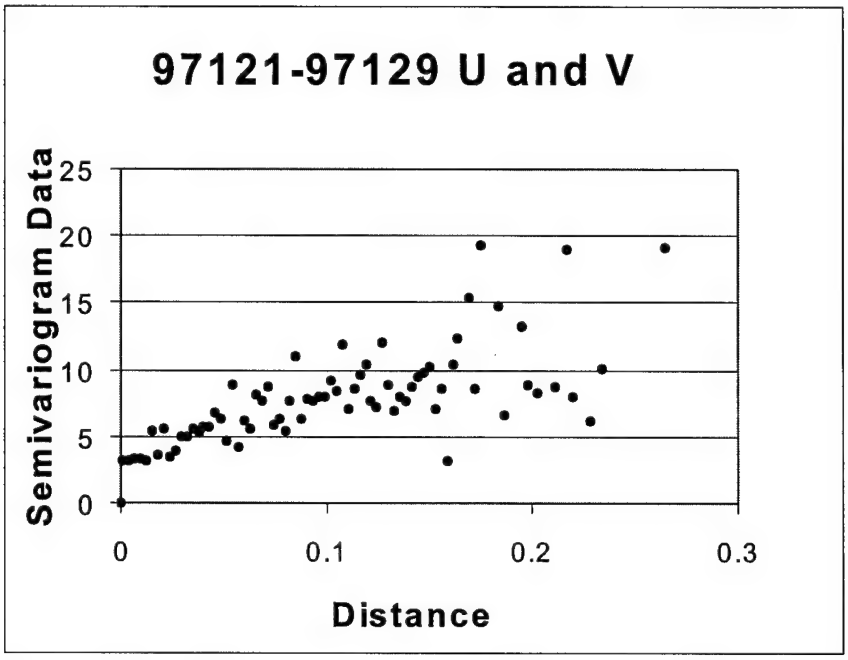

Figure 7 – Semi-variogram (9 Day Average)
This graph shows the semi-variogram data averaged over the 288 time slices for each of the Julian days 97121 through 97129. The units of distance are degrees squared and the units for the semi-variogram data are knots squared.

Once this method of averaging demonstrated its potential, the averaging process was started over. The days used to calculate the semi-variogram were between the Julian dates of 121 and 273 of 1997. The summer months were selected to avoid the dynamic weather patterns associated with the synoptic systems that impact the region during the winter months. It was hoped that with each new day's worth of data being added that a point would be reached at which the scatter-gram would stop changing. This stabilization did occur until the ten-day period from 97170 to 97180 was added. It was determined that the data started becoming more variable as the peak thunderstorm period was entered.

In an effort to get the best representation of the variability of the wind values it was determined that all the data for the time period, Julian dates 97121 through 97273, should be used to create the semi-variogram data. This was accomplished resulting in a total of 39,112,633 data pairs being used to produce Figure 8.

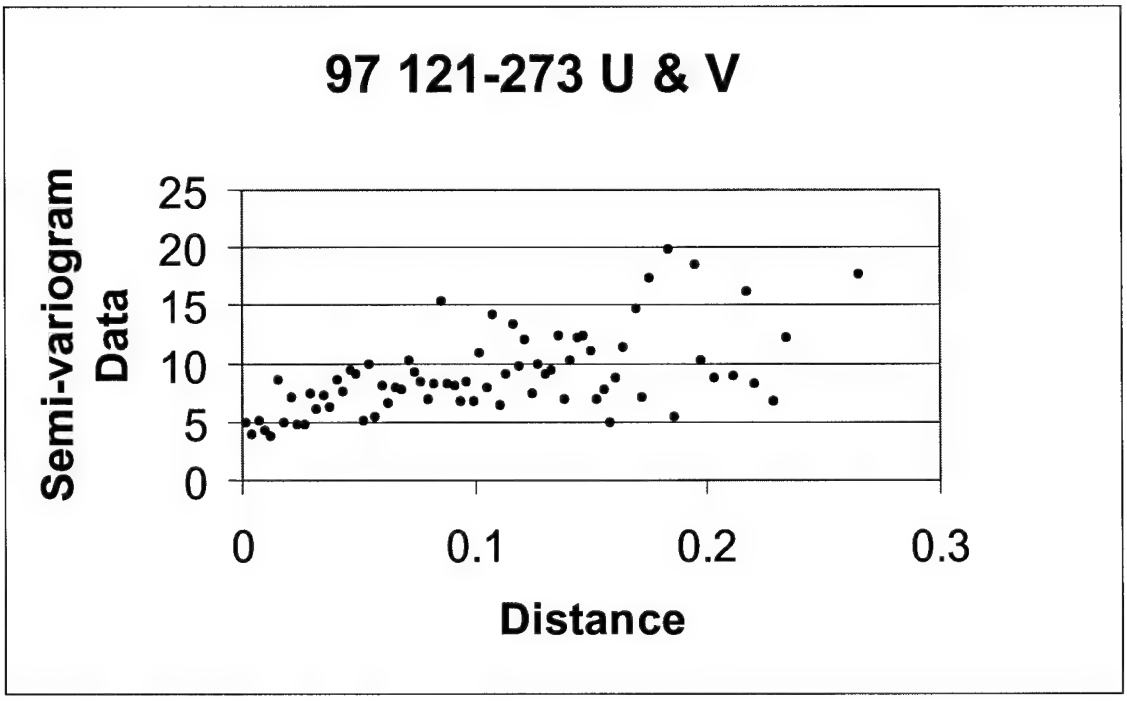

Figure 8 - Semi-variogram (97121-97273)
This graph shows the semi-variogram data averaged over the Julian dates 97121 through 97273. The units of distance are in degrees squared and the units of semi-variogram data are in knots squared.

The next task was to determine what model to fit to the semi-variogram data. Most of the literature reviewed addressed the subjective nature of fitting a model to the data. It was determined that allowing the computer to find the best-fit

functions would result in a much more accurate fit. The trendline function in Microsoft Excel was used to fit both a 2nd order polynomial and a 3rd order polynomial to the data. The difference between the two fits was minimal, as shown in Figure 9. Due to the minimal improvement of the higher order polynomial, the 2nd order polynomial was used, which allowed for a less complicated match to one of the standard models used for Kriging.

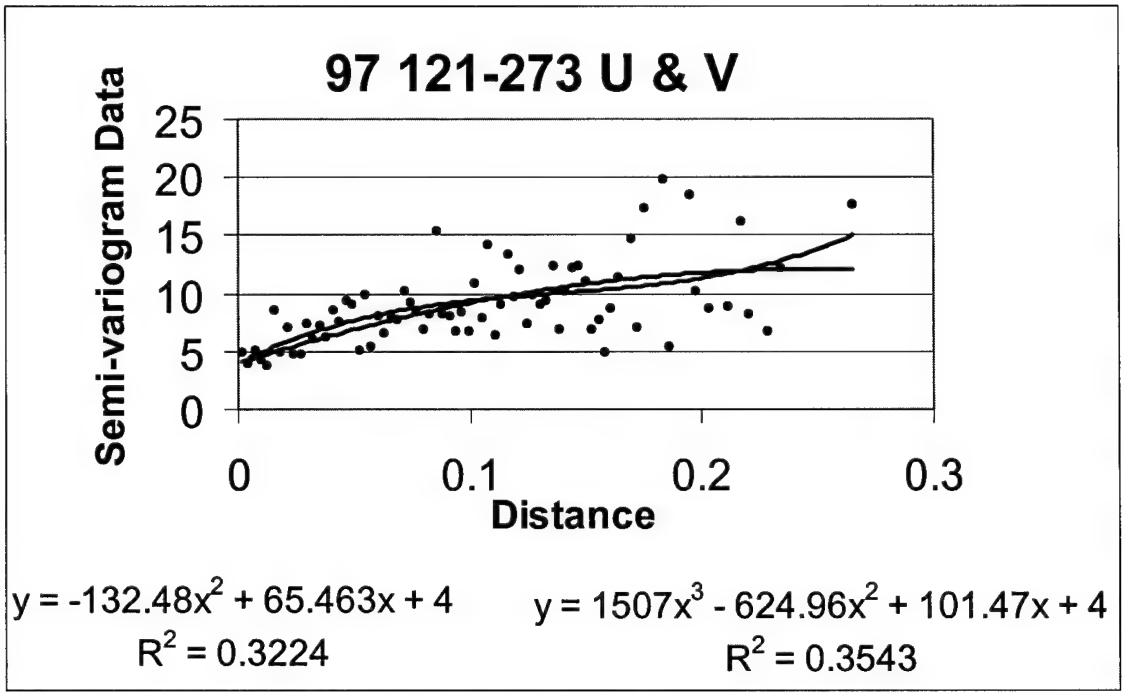

Figure 9 – Semi-variogram w/Polynomial fit
This figure shows the semi-variogram data averaged over the Julian dates 97121-97273. The units of distance are in degrees squared and the units for the semi-variogram data are knots squared. The computer generated best-fit 2nd and 3rd degree polynomials are plotted. The R^2 value shows the small improvement in the accuracy of the fit with the increased order of polynomial used.

The match was accomplished by graphing the polynomial, the spherical model and the exponential model. The parameters for the spherical and

exponential models were adjusted until the graphed line matched the computer generated 2nd order polynomial as closely as possible. Figure 10 shows how close the following exponential function matches the 2nd order trendline:

$$\gamma(h) = 4.3 + 9.9\left(1 - \exp\left(-\frac{3h}{0.44}\right)\right)$$

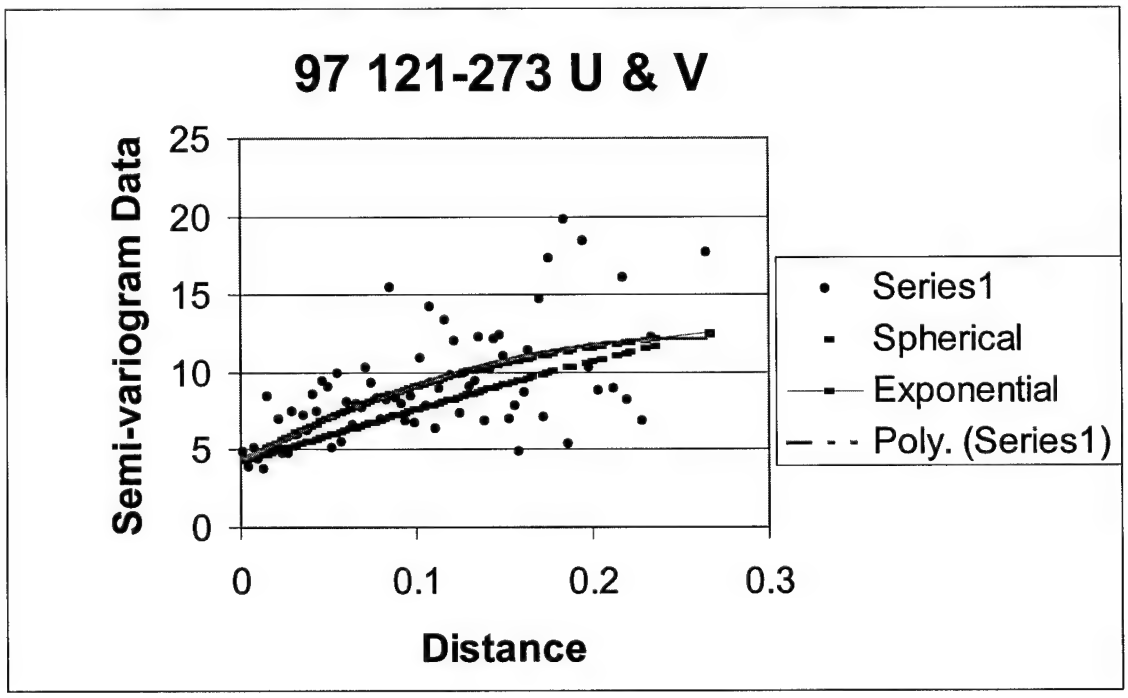

Figure 10 – Semi-variogram 97121 – 97273
This graph shows the fit of the exponential model and the spherical model when compared to the best-fit 2nd degree polynomial. The exponential model matches the polynomial curve almost perfectly and the spherical model falls below both of them. Units of distance are degrees squared and the units for the semi-variogram data is knots squared

The line representing the exponential model matches the 2nd order polynomial so closely that the two lines are hard to distinguish from each other. The lower, and straighter, line is the spherical model. Based upon the fit of the exponential model to the computer generated "best fit" 2nd order polynomial function, the exponential model was selected to be used for the semi-variogram model. The parameters for this function were then used to produce the covariance model. This covariance model had the following parameters, which were taken directly from the exponential semi-variogram model that was fit to the data:

$$C_0 = 4.3$$
$$C_1 = 9.9$$
$$a = 0.44$$

and had the form,

$$\tilde{C}(h) = \begin{cases} 9.9 \exp\left(-\dfrac{3h}{0.44}\right) & \text{if } h < a \\ 14.2 & \text{if } h \geq a \end{cases}.$$

This model was used in the computer program listed in Appendix A to create the ***C*** and ***D*** matrixes.

5. Vertical Interpolation

The selection of the vertical interpolation method was driven by the desire to keep the data smooth in the vertical. In general, wind directions and wind speeds do not display a discontinuity. Even in the event of a frontal boundary or

a temperature inversion the winds do not abruptly change when viewed at a fine enough scale. The selected method was cubic splines, which offers several benefits over using linear interpolation. The use of the cubic spline also has advantages over lower order splines.

Splines provide a polynomial fit between data points (knots) with certain smoothness criteria (Cheney and Kincaid, 1985:258). A first-degree spline produces a straight line between the data points. A second-degree provides a curve where the function and its first derivative are continuous at the knots. The cubic spline, 3rd-degree spline, has the added criterion of the second derivative also being continuous. The cubic spline produces a line that does not change abruptly at each knot and is smooth to the eye (Cheney and Kincaid, 1985:269). According to Cheney and Kincaid (1985) the use of higher degree splines seldom provides a greater advantage; they conclude the cubic spline is the best interpolating function. The implementation used here to solve the cubic spline closely follows their algorithm.

6. Error Estimation

This section is the heart of this work. Without some method of measuring the error associated with each of the methods tested it would be impossible to distinguish between their performance. Isaaks and Srivastava (1989) point out that estimation models are neither right, nor wrong, without external information to test them against. Without that external data, the best one can do is try to

determine whether the model is appropriate or not (Isaaks and Srivastava, 1989:198). This being the case, how does one determine how "appropriate" one model is over another? Any arbitrary method could have been used to provide numbers for the grid points. The object was to find one that would provide the most accurate estimation of the true wind. While the actual wind was not known everywhere, it was known at the observation locations. The logical method to determine how well the objective analysis algorithms estimate the wind was to withhold an observation and then compare that observation to the estimated wind at that same point. The difference would be the error associated with that estimate.

This was the method used during the evaluation of the two methods. The selection of which observation to withhold was accomplished by using a random number generation function available in the C++ standard library. This function was used to generate a random number between 0 and 100. Three hundred numbers were written into a data file. These numbers were then read in during the program operation to ensure that both the Kriging and Barnes method were evaluated with the same sequence of stations being withheld. When this number was read into the program, the modulus was calculated using the number of sensors available. This resulting number was then used as an index to determine the sensor data that was withheld.

For each time slot one sensor was withheld. The data associated with that sensor and the timeslot was also recorded. Once the objective analysis was performed and the estimated wind field had been calculated, the estimated wind

value was found by performing linear interpolation using the four grid points surrounding the wind tower location. Then the following information was written to a data file: the wind sensor identification number, the observed *u* component of the wind, the observed *v* component of the wind, the average observed *u* component, the average observed *v* component, the standard deviation of the observed *u* component, the standard deviation of the observed *v* component, the estimated *u* component of the wind, the estimated *v* component of the wind, the average estimated *u* component, the average estimated *v* component, the standard deviation of the estimated *u* component, and the standard deviation of the estimated *v* component. One data file was produced for each day evaluated. The information listed above was recorded for each time slice.

The first test was performed on a data file containing a uniform wind field. This provided a controlled test case where the estimated results were known beforehand. Both of the objective analysis methods being evaluated calculate the estimation based upon a weighted summation of the observations. If all the observations are the same, then the estimated value must also be the same as the observations. This test provided a means of checking the accuracy of the algorithms to ensure they performed as expected in the simple idealized case.

After this test was performed, one day was randomly selected from each of the compressed archive files for a total of 16 days and 4535 time slices being evaluated. The selected days and the number of time slots in each day is provided in Table 1. Those days not containing all possible 288 time slices are identified with an asterisk.

Julian Date	Number of Time Slices	Julian Date	Number of Time Slices
97128	288	97208	288
97134*	250	97217*	283
97148	288	97222	288
97159	288	97234	288
97166	288	97245	288
97170*	258	97253	288
97183	288	97267	288
97191	288	97270	288

Table 1 – Tested Days and Number of Time Slices
This table provides the Julian dates of the days that were tested and the number of time slices each of those days contained.

7. Summary

This chapter provided a detailed record of the steps followed during the implementation of the objective analysis methods. A brief description of the computer program functional design provided the basic program flow. The flow followed this general pattern: read in all of the observation data, calculate the estimated field on the horizontal planes, perform vertical interpolation, and output

the values in a vis5d format. What followed was a discussion of the data integration and then a brief explanation of how the 4k parameter for the Barnes method was dynamically calculated during program execution. The major portion of the chapter was dedicated to the description of how Kriging was implemented.

IV. Results Analysis

1. Chapter Overview

This chapter provides a detailed record of the data collected during the test phase of the implementation. It also provides insight as to what the data means about the accuracies of the estimation methods. Most importantly, the data is used to conclude which of the two objective analysis methods provides the most accurate estimation of the wind field and if that estimation is accurate enough to base a visualization program on.

2. Results

The first test performed was on the uniform wind field data set. The wind was set at 335 degrees and 2 knots for all of the reporting stations. This produced a U component of 0.845 and a V component of -1.813. Both Barnes and Kriging produced essentially these same values for their estimates. Table 2 shows that the observed value minus the estimated value yielded an error that was negligible. The standard deviation for both methods was extremely small. These results met with expectations and showed that both of the methods were producing the proper estimated values.

Barnes error - U	Barnes error - V	Kriging error - U	Kriging error - V
1E-06	-1E-06	0	1E-06
Observed Barnes STD - U	Observed Barnes STD - V	Observed Kriging STD - U	Observed Kriging STD - V
0	0	0	0
Estimated Barnes STD - U	Estimated Barnes STD - V	Estimated Kriging STD - U	Estimated Kriging STD - V
0.000002	0.000005	0.000002	0.000005

Table 2 – Uniform Wind Field Results
This table provides the accuracy details about the test performed on the uniform wind field. The errors were calculated by subtracting the estimated value from the observed. Barnes and Kriging performed with almost equal accuracy.

The next set of tests was performed on each of the randomly selected data files. Each of these files contained one day's worth of wind observations. The selected days are provided in Table 3. To determine how well the estimated wind value matched the observed value, the correlation between the two data sets was calculated. If there was no error, the estimated wind would match the observed wind and the correlation would be 1 (Mendenhall and Sincich, 1992:214).

The correlation numbers provided in Table 3 show that for every day evaluated except two the estimated values produced by Kriging correlated better to the observed values than did those estimated with Barnes' method. The average correlation for the U component using Barnes was almost 0.5 while Kriging had a correlation of 0.55. For the V component of the wind the improvement of the Kriging estimate over the Barnes estimate was even larger. Kriging's correlation was nearly 0.34, while the Barnes correlation was only 0.25.

Date	Barnes Correlation - U	Barnes Correlation - V	Kriging Correlation - U	Kriging Correlation - V
97128	0.238	0.063	0.408	0.154
97134	0.512	0.332	0.578	0.414
97148	0.871	0.309	0.877	0.420
97159	0.594	0.546	0.679	0.514
97166	0.677	0.556	0.702	0.626
97170	0.480	0.117	0.543	0.144
97183	0.674	0.490	0.700	0.585
97191	0.250	-0.068	0.313	0.050
97208	0.003	0.318	0.039	0.452
97217	0.701	0.181	0.671	0.181
97222	0.521	0.240	0.596	0.383
97234	0.706	0.178	0.709	0.343
97245	0.291	0.160	0.390	0.379
97253	0.622	0.358	0.669	0.445
97267	0.214	-0.099	0.339	-0.002
97270	0.610	0.298	0.611	0.338
Average Correlation	0.498	0.249	0.551	0.339

Table 3 – Estimated/Observed Correlation
This table provides the results of the correlation between the estimated wind component and the observed wind component for both Kriging and Barnes.

The range of correlation values was from –0.099 to 0.877. This means that some days the estimation schemes worked very well and others it did not work at all. To provide a better understanding of what these correlation numbers mean, close consideration is given to the day with the best correlation, the day with the worst correlation, and a day that represents the average case.

a. Best Case

The best case correlation occurred on day 97148 with a correlation value of 0.871026 for Kriging on the *u* component of the wind. There was no significant synoptic weather event associated with this day. The land/sea breeze is evident in the *u* component of the wind by the positive and negative groupings of the wind. Figure 11 shows a plot of the estimated values versus the observed values. Also plotted on the graph is a line that shows the line the data would fall on if there was zero error. The majority of the data falls within a band that is within 3-4 knots of the error free line.

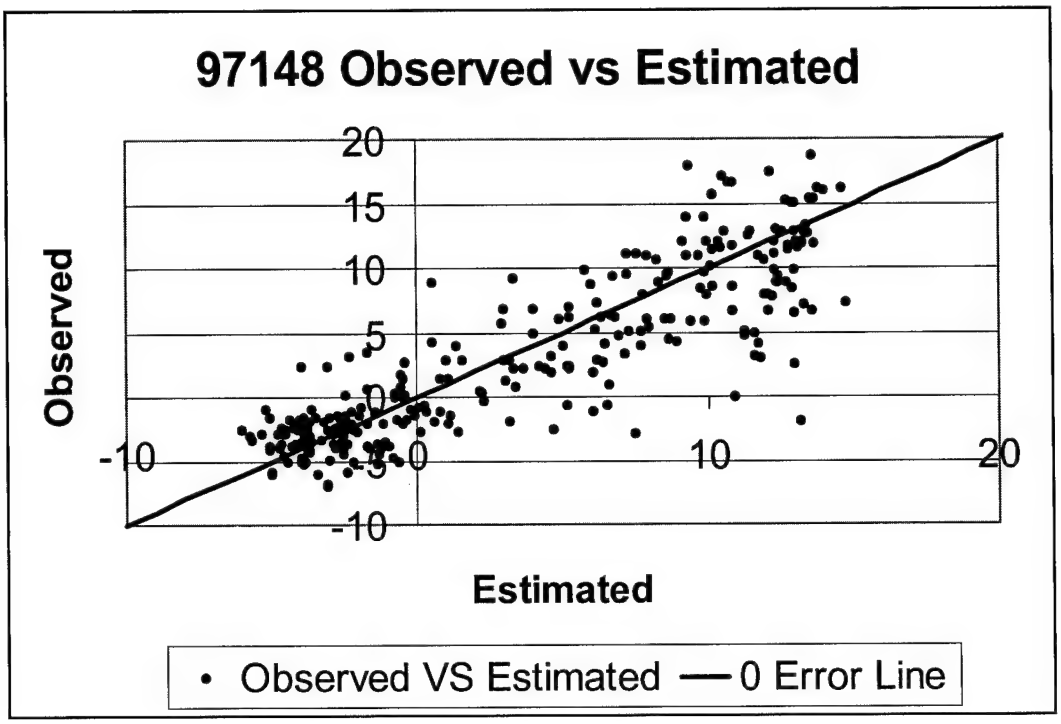

Figure 11 – 97148 Observed vs Estimated
This figure depicts the scatter plot of the observed *u* component of the wind in knots versus the estimated wind for the Julian day 97148. The solid line represents a line of 0 error.

The average absolute value of the error for the *u* component of the wind was 2.39 knots. The maximum magnitude of error was 15.05 knots and the minimum was 0.001 knots. This shows that the estimate can be very good or extremely poor.

Figure 12 shows the histogram data concerning the error magnitude, and Table 4 shows the cumulative percentage data. This table shows that over 73 percent of the errors fell within 3 knots and 88 percent of the errors fell within 5 knots.

Error Magnitude(Knots)	Frequency	Cumulative Percentage
0	0	.00%
1	81	28.22%
2	76	54.70%
3	54	73.52%
4	28	83.28%
5	14	88.15%
6	13	92.68%
7	8	95.47%
8	5	97.21%
9	4	98.61%
10	0	98.61%
11	3	99.65%
12	0	99.65%
13	0	99.65%
14	0	99.65%
15	0	99.65%
> 15	1	100.00%

Table 4 – 97148 Cumulative Histogram Table
This table provides the percentage of the values that occur at or below the specified error magnitude.

The average observed *u* component of the wind was 3.2 knots while the average Kriging estimated *u* component was 3.4 knots. The average observed *v* component was 3.0 knots, with the average Kriging estimated *v* component being 3.5 knots. Both the estimated *u* and *v* components of the wind were larger than the observed wind. The standard deviations (std) show the following: the observed std of the *u* component was 2.9 and the std of the observed *v* component was 2.7 knots. These values are larger that the std of the estimated components. The std of the estimated *u* component is 1.7 knots and 1.5 knots for the estimated *v* component. These numbers show that Kriging produces an

overestimate of the wind field, but has a reduced variability. This reduction in the variability represents a smoothing of the data.

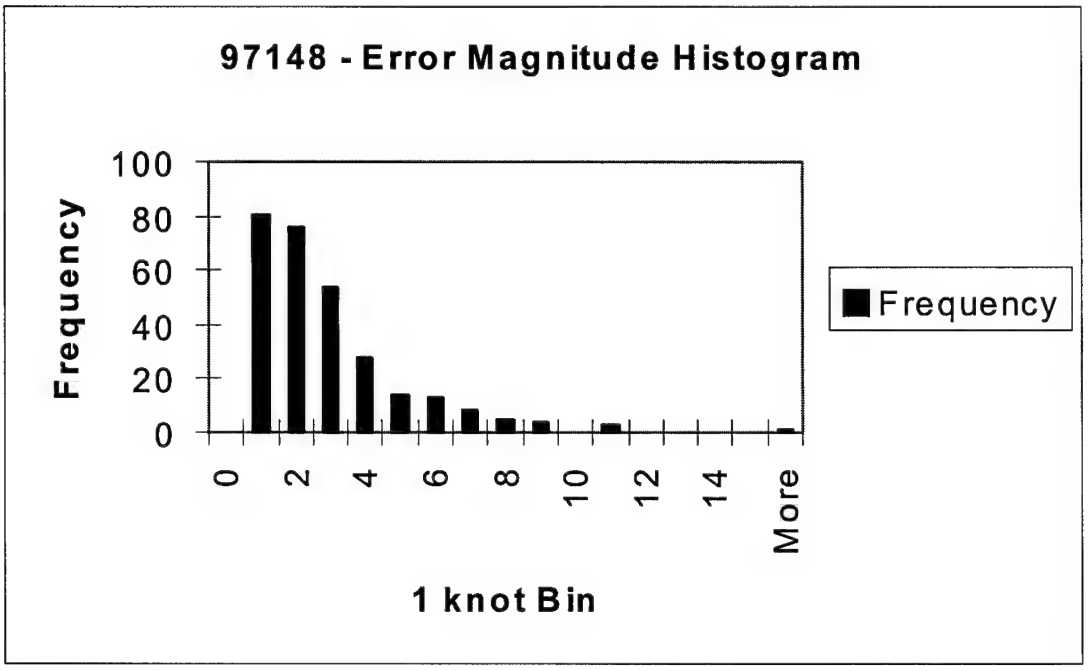

Figure 12 – Error Magnitude Histogram
This histogram shows the frequency of errors measured in 1-knot increments.

b. Worst Case

The next task is to consider the worst case day, 97267. This day had a Kriging correlation for the *v* component of the wind of -0.002. The lack of positive values is evidence of the synoptic scale forcing, which was strong enough to dominate the land breeze.

Figure 13 shows a plot of the estimated values versus the observed values. Also plotted on the graph is a line that shows the line the data would fall on if there was zero error. Unlike Figure 11 the data in this graph does not group closely around the 0-error line. This is a visual confirmation of why this day is characterized by a low correlation.

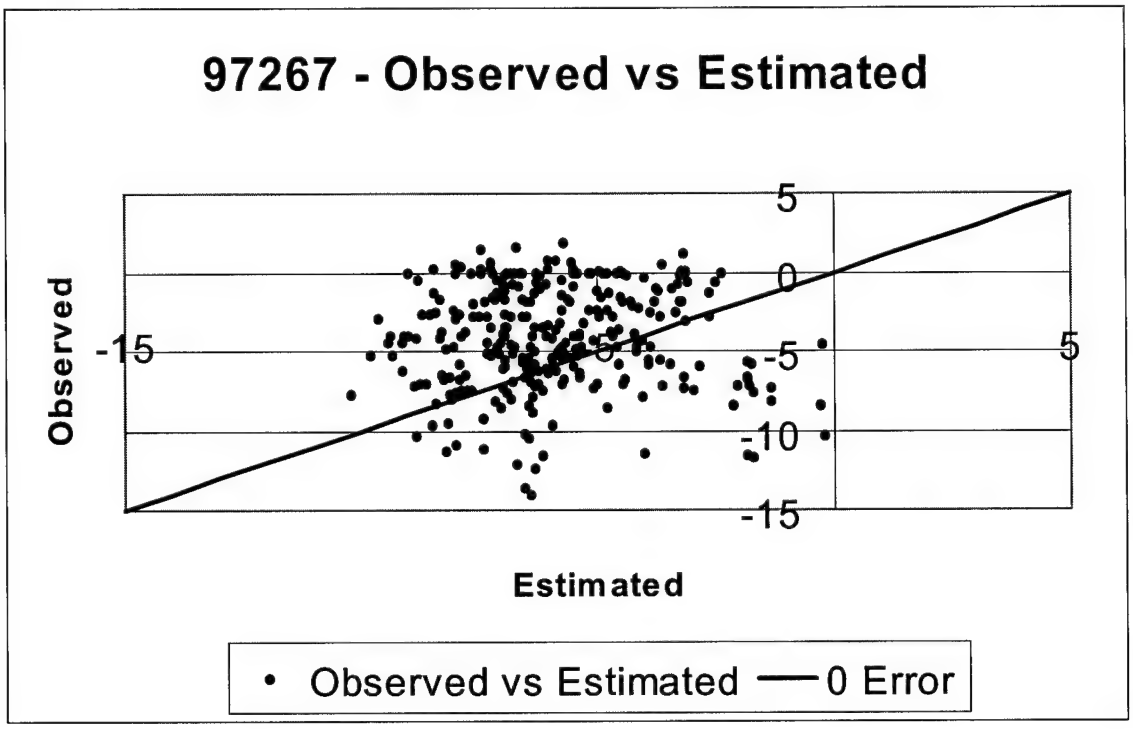

Figure 13 – 97267 Observed vs Estimated
This figure depicts the scatter plot of the observed *v* component of the wind in knots versus the estimated wind for the Julian day 97267. The solid line represents a line of 0 error.

The average absolute value of the error for the *v* component of the wind was 3.4 knots. The maximum magnitude of error was 10.1 knots and the minimum

was 0.01 knots. Figure 14 shows the histogram data concerning the error magnitude, and Table 5 shows the cumulative percentage data. This table shows that the maximum error might be less than that in the best case scenario, but there are a larger number of errors that are greater than three knots. In the 97148 case over 73 percent of the errors fell within 3 knots, but for the worst case day only about 50 percent of the errors fall within 3 knots.

Error Magnitude (Knots)	Frequency	Cumulative Percentage
0	0	.00%
1	54	18.75%
2	49	35.76%
3	39	49.31%
4	31	60.07%
5	36	72.57%
6	33	84.03%
7	21	91.32%
8	14	96.18%
9	7	98.61%
10	3	99.65%
11	1	100.00%
> 11	0	100.00%

Table 5 – 97267 Cumulative Histogram Table
This table provides the percentage of the values that occur at or below the specified error magnitude.

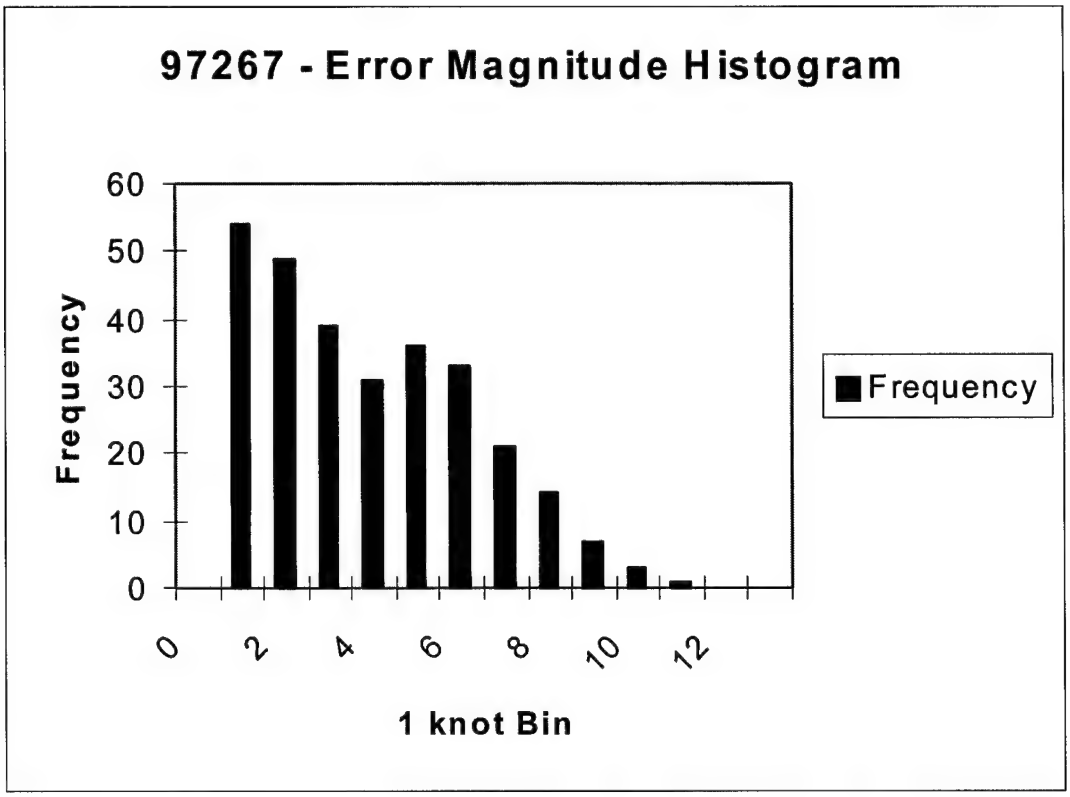

Figure 14 – 97267 Error Magnitude Histogram
This histogram shows the frequency of errors measured in 1-knot increments.

c. Average Case

Finally, a case representing the average case is considered. The selected day has a correlation similar to the average *u* wind component correlation for the Kriging method. The day 97134 had a correlation of about 0.58, which is close to the 0.55 average.

Figure 15 shows a plot of the estimated *u* values versus the observed values. Once again the line representing 0 error is also plotted on the graph.

Much of the data does plot close to the 0-error line; however, as the data becomes more positive it begins to deviate from that line much more.

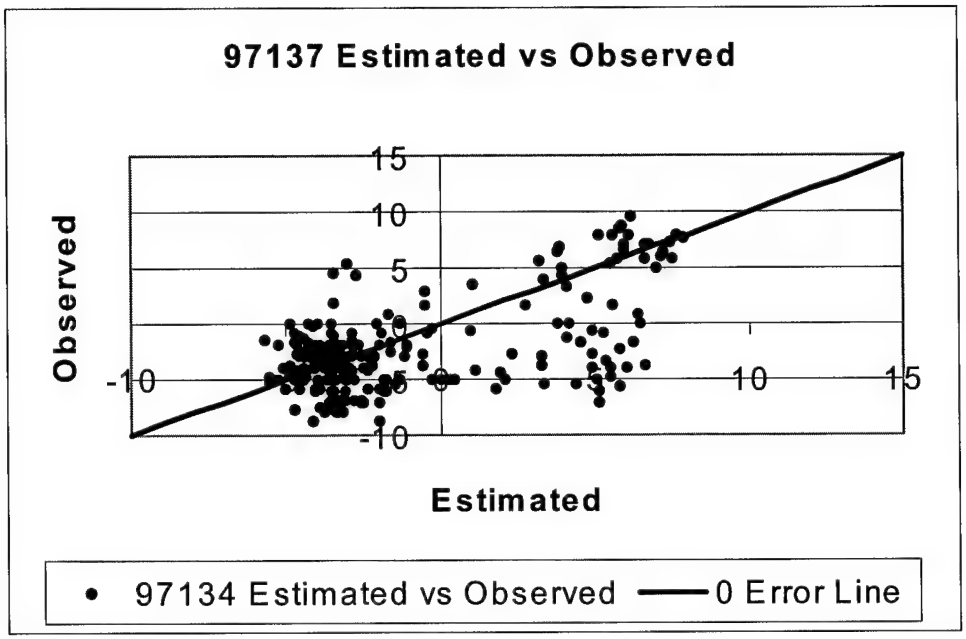

Figure 15 – 97137 Estimated vs Observed
This figure depicts the scatter plot of the *u* component of the observed wind in knots versus the estimated *u* component of the wind for the Julian day 97137. The solid line represents a line of 0 error.

The average magnitude of the error for the *u* component of the wind was 2.7 knots, and the average magnitude of the error for the *v* component of the wind was 1.3 knots. The largest magnitude of error for the *u* component was 12.1 knots and the minimum was 0.03 knots. The largest error for the *v* component was 6.0 knots, and the minimum was 0.001 knots. Once again the standard deviations were significantly reduced indicating the degree of smoothing the estimation introduces into the estimated wind field.

Figure 16 and Table 6 provide the error magnitude distribution information. Comparing this information to the best and worst case days based upon the correlation numbers supports the correlation results. Looking at the percentage of errors that have magnitude of three knots or less yields 73 percent in the best case, almost 50 percent in the worst case, and 66 percent for this day representing the average case.

Error Magnitude (Knots)	Frequency	Cumulative %
0	0	.00%
1	85	34.00%
2	53	55.20%
3	27	66.00%
4	28	77.20%
5	19	84.80%
6	10	88.80%
7	6	91.20%
8	7	94.00%
9	6	96.40%
10	3	97.60%
11	3	98.80%
12	2	99.60%
13	1	100.00%
> 13	0	100.00%

Table 6 – 97134 Cumulative Histogram Table
This table provides the percentage of the values that occur at or below the specified error magnitude.

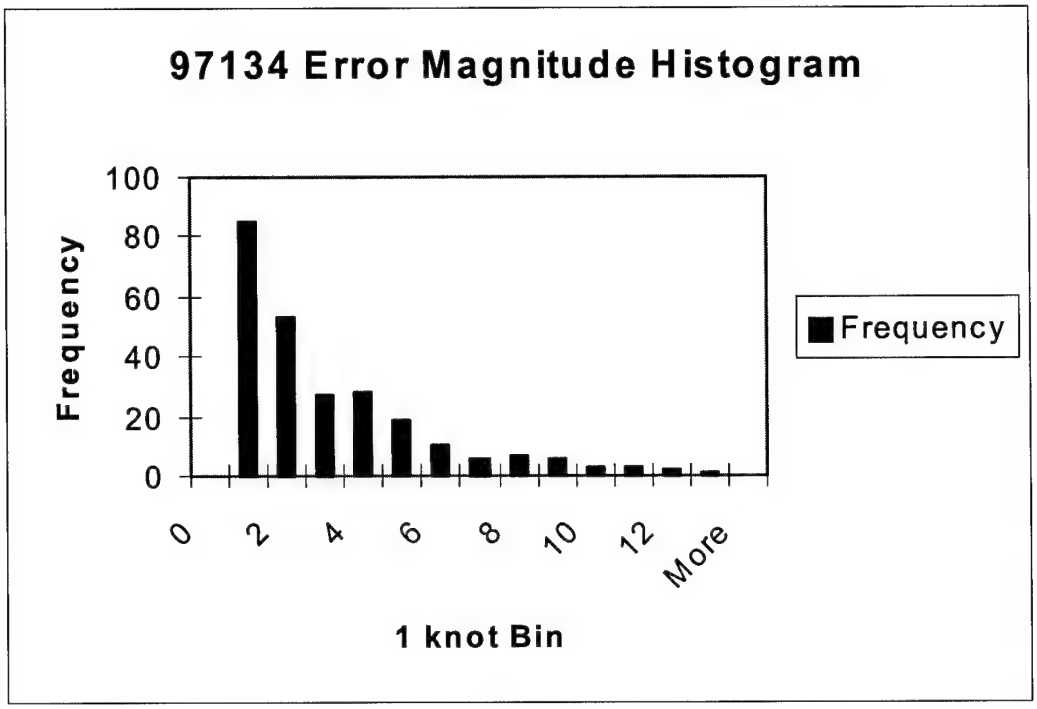

Figure 16 – 97134 Error Magnitude Histogram
This histogram shows the frequency of errors measured in 1-knot increments.

d. Error as a function of station

Due to the importance of distance upon the objective analysis methods evaluated, one might suspect that the error associated with an estimate would be a function of how far it is from the observations the estimate is based upon. The assumption would be that points farther from the observations, or outside the domain of the observations, would have the largest error associated with them.

To evaluate this relationship, the data collected from each day tested was inspected to determine which wind towers had the largest magnitude of U and V

error associated with them. Stating the error is associated with a tower does not indicate the observation was incorrect. The wind tower locations are the only places the error is known, because certain observations were withheld for the purpose of evaluating the "true" value against the estimated.

The number of times each tower appeared was recorded. Then the same process was accomplished again, but the tower IDs were recorded if they had one of the largest five errors. The five tower identification numbers that appeared the most times on each list were compared.

Table 7 shows the results of this evaluation. There was a total of six wind tower IDs that appeared in the two lists.

	1st	2nd	3rd	4th	5th
Most-Top 1	1617	819	3	1612	421
Most-Top 5	1617	819	1612	2016	3

Table 7 – Wind Towers with Largest Estimation Error
This table lists the towers that most frequently had the largest error associated with them on the first row (1st indicating the most frequent). On the second row is listed the towers that most frequently had one of the largest five errors. Note that four of the towers appear in both lists.

Figure 17 shows the number of occurrences for all of the wind towers that had one of the top five errors. The five towers that most frequently had one of the top five errors accounted for 66 percent of these errors. The same was true when the data considering only the top error was evaluated. These results show that the location of where an estimate is calculated is an important factor in determining the degree of error associated with that estimate.

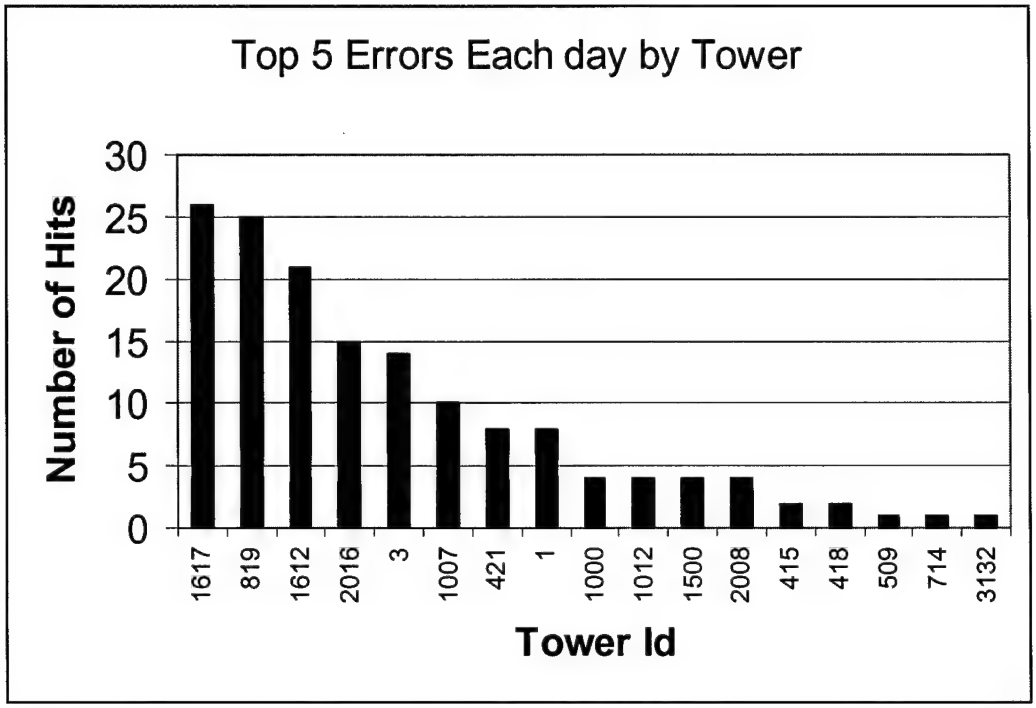

Figure 17 – Stations with Top Five Error Magnitude
This histogram provides the frequency that each tower had one of the largest five errors associated with it.

Looking at Figure 18 provides visual confirmation of the suspected relationship between the wind towers with the largest error and the remaining observations. In the figure, wind tower locations are marked with a small black circle. The location of the six wind towers identified in Table 7 as having the largest errors associated with them are identified by the larger gray circles. Figure 18 shows that these six stations are located on the outer edges of the area that bounds the entire set of observation locations. This was as expected.

The further the location of the estimated value got from the observations it was based on, the larger the error.

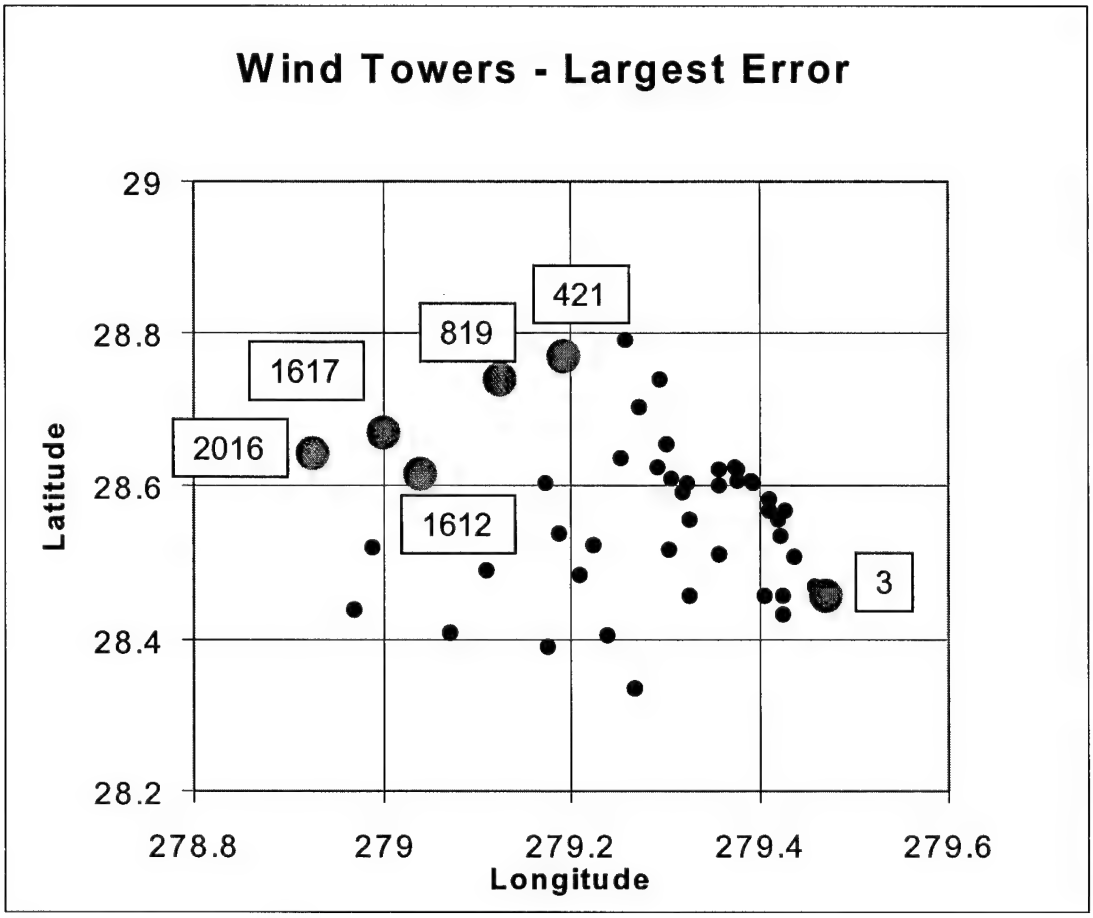

Figure 18 – Stations with Largest Error Magnitude
This figure shows wind tower locations where the greatest frequency of having the largest error, or one of the top five largest errors, was encountered.

3. Summary

This chapter presented the results of the testing. These results showed that Kriging provided a more accurate estimate than the Barnes method. Approximately 94 percent of the time the estimated value correlated better to the observed value for Kriging than Barnes. Also, about 93.3 percent of the time Kriging had a smaller value for the maximum magnitude of error. Even though Kriging provided a better estimate, it still produced errors that exceeded 10 knots. Another impact of the Kriging estimation method was that it smoothed the data considerably. Indeed, Kriging smoothed the data more than Barnes did. Finally, the relationship between the estimated value's location and the supporting observations' locations was examined. The result showed that the further apart they were the larger the error became.

V. Conclusions/Recommendations

1. Conclusions

The results presented in Chapter 4 support the conclusion that Kriging produced a more accurate estimated wind field than did Barnes' method. The correlation numbers were better for Kriging, and the average magnitude of the error associated with Kriging was lower than with Barnes' method. Therefore, if this project continues without further research, Kriging should be implemented as the objective analysis method.

It's important to understand that even though Kriging produced the best estimate of the wind field, it has its limitations. In the best case scenario almost 12 percent of the errors were greater than or equal to 5 knots. Obviously, the best case rarely occurs. Even if one can hope for the average case scenario happening all the time, almost 16 percent of the time the estimated wind component will have an error exceeding 5 knots. The only way to drastically improve this estimation is to increase the number of observations taken at all levels. This would require an increase in the number of wind towers as well as the addition of sensors at additional heights on the current towers. All the evaluations were performed at the height of 54 feet because there were as many as 35 observations available on which to base the estimate. Some of the other heights had only one or two sensors available.

2. Recommendations

There are three alternatives to consider. The first is to continue with this present objective analysis effort. The second is to use a mesoscale model to produce a wind field estimate. The third is to examine another method of graphically displaying the observations without performing the estimations.

The first option to consider deals with continuing the present course. This is not a recommend course of action. The errors associated with the estimation techniques evaluated here are significant enough to cast serious doubt on their ability to create a realistic three-dimensional wind field that can be used to more accurately forecast or observe the winds at Cape Canaveral. Of the methods considered here, Kriging produced the most accurate estimated wind field. If it is determined that the use of an objective analysis method is the proper course to follow, then it is recommended that the Bratseth method, or some other type of optimal interpolation, be evaluated to determine if it could provide a better estimate than Kriging.

The second option is the use of a mesoscale model, or some model that would include atmospheric dynamics. The use of a model would allow for the influence of local frictional and thermal properties to be accounted for. The concept is for the observations to be used as input. The model could then be run to fit a pressure field to the observed wind field. The derived pressure field could then be used to estimate the winds throughout the three-dimensional domain.

The third option would be to examine a method to graphically display the observations in a three-dimensional image. This display would not have the ability to isopleth wind values. It would simply display the observations in three-dimensional space with some type of symbol or object so that the observations can be visualized in proper physical relationship with each other. This option would allow for the data to be integrated into a single three-dimensional display. A variety of options could be used to display the magnitude and direction of the wind at each observation point, like a windsock. Its size or color could be altered to indicate magnitude. It could simply display the observed data in some convenient and coherent three-dimensional image.

3. Summary

In conclusion, the objective analysis methods have a large error associated with them and it is not recommended that they be used to estimate the low level wind field.

Appendix A: C Code for everything

```
// File: ObAnal.h
// Author:  Lt Mike Engel
// Date:
//
//  This header file contains the declairations required to perform the
//    the objective analysis of the CAPE area
//
// modified on 13 Nov 98 to add the Kriging class

#define MAX_LAT 29.0
#define MAX_LON 280.0
#define MIN_LAT 28
#define MIN_LON 278.8
#define MAX_ALT 3000
#define LAT_STEP 0.05
#define LON_STEP 0.05
#define ALT_STEP 48.0
#define NUM_SENSORS 40 // 50    // 9 22
#define NUM_HEIGHTS 10 //10
#define MAX_FILENAME_LEN 80
#define TRUE 1
#define FALSE 0
#define PI 3.14159265359
#define XRANGE  25 // 13 //(MAX_LAT-MIN_LAT)/LAT_STEP + 1
#define YRANGE  21 // 11 //(MAX_LON-MIN_LON)/LON_STEP + 1
#define ZRANGE  21 //MAX_ALT/ALT_STEP
#define K41 1      // for 1 call to the barnes sub routine
#define K42 0.3    // for the second call to the barnes sub
#define MAXVARS 5
#define MAXTIMES 288 // 72   // 50 really

int outputvert = 1;

//  the Wind class is used as one of the base classes for the Windfield
class
//    it provides the data structure for the actual wind data
//
class Wind {
   public:
      Wind(){u=-999.0; v=-999.0;};    // the constructor
       void setWind(float, float);
       void setWindV(float, float);    // vector version (degree/speed)
       void getWind(float *, float *);
       void getWindV(float *, float *); //vector version (degree/speed)
   private:
       float u;     // the u component of the wind
       float v;     // the v component of the wind
};
```

```
//  start the structures required for the Windfield class
//
struct sensorType {

   int id;          // this is the station id number
    float latitude;  // the lat and lon of the sensor location
    float longitude;
    float height;
   int sensorHeights[NUM_HEIGHTS];
};

struct obsDataType {
    Wind data[NUM_SENSORS][NUM_HEIGHTS];
    long date;
    long time;

};

struct Field3DType {
    Wind data[XRANGE][YRANGE][NUM_HEIGHTS];  // zrange ?
};

struct Int2DType {
    int index[NUM_HEIGHTS][NUM_SENSORS];
};

// one way linked list to allow multiple time steps for observations
struct obsListType {
    obsDataType *observation;    // pointer to the observation data
    obsListType *next;
};

class ObsWindField {
    public:
        ObsWindField(); // constructor for the
        ~ObsWindField();//destructor-frees the memory in the linked list
      void getSensorData();//open the data files containing the sensor
information
       int getWindData();//open the data files containing the observed
data
                    // read the data and return the number of time steps
       void collectData();
       void displayInput();
       void displaySensorInfo();
       void test();
      void newObsList();
       obsDataType *currObs;           // observed wind values --
public to avoid lots of space
       obsListType *obsListHead;
      obsListType *obsList;
       sensorType sensorInfo[NUM_SENSORS];    // data concerning sensor
location
      int getSensorIndex( long );
       int getHeightIndex( long );
    private:
```

```
};


class Barnes {
    public:
        Barnes();           // constructor for the Barnes class
        ~Barnes();
        void doBarnes();    // main function controlling the analysis
        int output2dAnalField( char *);        // print to a vis5d file
        int output2dAnalFieldHdr( char * );
        void displayAnalField();    // print anal field to the screen
        void getAnalData( int n, int xrange, int yrange );
                    // collect data on the anal field
    private:
        void barnesSub(float,int,int,int,int);
                // subroutine performing the analysis for each iteration
        float dist( float, float, float, float );
                // function returing distance in
                // degrees square
        ObsWindField obsWind;
        obsDataType *corrFactor;
        Field3DType *anal1;
        Field3DType *anal2;
        Int2DType *sensor;
        // int levelIndex[1][1];
        // int levelIndex[NUM_HEIGHTS][NUM_SENSORS];
        int numOfSensors[NUM_HEIGHTS];
        int numOfTimeSteps;
};


// start the class definition for the kriging class
//   this will contain all of the data elements and the functions
// required to perform kriging
class Kriging {
    public:
        Kriging();          // constructor for the Kriging class
        ~Kriging();         // destructor for the Kriging class
        void doKriging();   // main function controlling the analysis
        int output2dAnalField( char *);   // print data to a pass1 file
        int output2dAnalFieldHdr( char * );
                    // print header information to the pass1
        void displayAnalField();    // print anal field to the screen
        float Kriging::coVariance( float dist );
                    // return the coVariance value
        void getAnalData( int n, int xrange, int yrange );
                    // collect data on the anal field
    private:
        void KrigingSub(int,int,int);
            // subroutine performing the analysis for each iteration
        float dist( float, float, float, float );
           // function returing distance in degrees square
        float semiVarioFunc( float );
        ObsWindField obsWind;
        Field3DType *anal1;
```

```
        Field3DType *anal2;
        Int2DType *sensor;
        // int levelIndex[1][1];
        // int levelIndex[NUM_HEIGHTS][NUM_SENSORS];
        int numOfSensors[NUM_HEIGHTS];
        int numOfTimeSteps;
};

int heightIndex[NUM_HEIGHTS] = {12,30,54,60,90,162,204,295,394,492};
int interpHeightIndex[ZRANGE] = {12,36,60,84,108,132,156,180,204,
        228,252,276,300,324,348,372,396, 420,444,468,492};


void cubicSpline( int n, float * t,float * y,float * h,float * b,
                      float * u,float * v,float * z );
void doVerticalInterpolation( void );
float InterpSpline( int n, float * t,float * y,float * z, int x );


// here are some global variables used to collect statistical data
int display = 0;
int dataCollection = 0;
int idList[MAXTIMES];
Wind actual[MAXTIMES];
Wind mean[MAXTIMES];
Wind std[MAXTIMES];

void callKriging( void );
void callBarnes( void );

// File: ObAnal.cpp
// Author:  Lt Mike Engel
// Date: 9 Oct 98
//
//  This file contains the code required to perform the objective
//  analysis of the observed data.
//
// include files
#include "obanal.h"
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <math.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <alloc.h>
// #include "matrix.h"

//  start with the functions associated with the Wind class
//  The following functions are the get and set functions for the
//     Wind class.  There are functions that deal with the u,v
//     components
//     of the wind as well as ones that use the speed and direction
```

```
void Wind::setWind( float newu, float newv) {
   u = newu; v = newv;
}
void Wind::setWindV( float dir, float speed) {
    u = speed*sin( (double) dir*PI/180 );
    v = speed*cos( (double) dir*PI/180 );
}

void Wind::getWind( float * retu, float *retv) {
   *retu = u; *retv = v;
}
void Wind::getWindV( float * dir, float * speed) {
   *speed = sqrt( u*u + v*v );
    *dir =  atan( (double)u/v)*180;    // not correct
}


//  start the class member functions for the ObsWindField class
//     Function:  getWindData
//      This function reads in the data from the data file.
//      It prompts the user for the data file name
//      It allows multiple data files to be merged

//  Function:  ObsWindField
//  This is the constructor function for the ObsWindField class.  This
//     allocates memory for the observations and initializes the field
//     to -999 for both the u and v components.

ObsWindField::ObsWindField() {

    obsListHead = NULL;

}

// Function: ~ObsWindField
//  This function is the destructor function for the linked list of
//  observation data. It checks to make sure there is data on the list
//  before free(). This is done in the while condition.
ObsWindField::~ObsWindField() {

    obsListType * step;
    obsListType * save;

    step = save = obsListHead;
    obsListHead = NULL;

    while( step != NULL ) {
       save = step->next;      // point to the next node
        free( step->observation );   // free the observed data
        free( step );                // free the time step node
        step = save;
    }

}
```

```
// function: newObsList
// This function allocates the memory required for the observation
//   data for a new time step
void ObsWindField::newObsList() {

    // first allocate memory for the next link in the obs chain
    if( obsListHead == NULL ) {
        obsListHead = (obsListType *) malloc(sizeof(obsListType));
        obsList = obsListHead;
    }
    else {
    obsList->next = (obsListType *) malloc(sizeof(obsListType));
    //step to that next link
        obsList = obsList->next;
    }
    // set the next pointer to NULL -- terminate the linked list
    obsList->next = NULL;

    // allocate memory for the obs data
    obsList->observation = (obsDataType *) malloc(sizeof(obsDataType));
    // currObs = (obsDataType *) malloc(sizeof(obsDataType));

//  set the pointer for the current obs being worked on to the init list
    currObs = obsList->observation;

    //allocate memory for the far pointer
    if( obsList == NULL || currObs == NULL ) {
          cout << "Insufficient memory to run" << endl;
          exit(1);
    }

    // intialize the fields to -999
    for( int i = 0; i < NUM_SENSORS; i++ ) {
       for( int j = 0; j < NUM_HEIGHTS; j++ ) {
            currObs->data[i][j].setWind( -999.0, -999.0 );
       }
    }
}

// function:  getWindData
// This function gets a file name from the useer and then opens the file
//    It reads in the observations and places them in the appropriate
//    array position depending upon the sensor id and the height.  If
//    the sensor id is not recognized then the data is ignored.

int ObsWindField::getWindData() {
   int done = 0,idindex = 0, heightindex = 0;
    long date,time = 1,lastTime = -1,oldtime,id,height,direction;
    float speed,gust,ddev,temp,td,rh;
   char filename[MAX_FILENAME_LEN],c1,c2,c3;
   char inbuffer[80],tempbuffer[8],yesno;
   char *errRet,*inputptr,*tempptr;
    FILE * indatafile;
   int t = 0;
```

```
// begin the outer loop -- allows multiple files to be read in

while( !done ) {

   //  allow user input of multiple data files
   // get the input
   cout << "Please enter the file name to open.\n";
   cin >> filename;
   cout << filename << " ";

   // open the file and begin to read it

   indatafile = fopen( filename, "r" );

   // check to make sure the file opened
   if( !indatafile ) {
      cout << "File could not be opened" << endl;
      exit(1);      // exit loop and program
   }

    // read in the header information
   for( int i = 0; i < 12; i++ )
      fscanf( indatafile, "%s ", &inbuffer);

   // now read in the data
    while ( fgets( inbuffer, 80, indatafile ) ){
      //first read in the date, time,id and height
      sscanf( inbuffer, "%7ld %6ld %4ld %c%c%c %5ld",
            &date, &time, &id,&c1,&c2,&c3,&height);

        if( time != lastTime ) {
            if( t >= MAXTIMES )
               break;   // exit the loop -- vis5d only allows 50
            newObsList();
            currObs->date = date;
            currObs->time = time;
            t++;
            cout << "TIME: "<< time << " " << t << endl;
      }
      direction = speed = gust = -1;
      ddev = temp = td = rh = -1;

      inputptr = &inbuffer[30];

      // the following data may or may not be in the file
      // read in the direction
      if( *inputptr != NULL && *inputptr != 10 ) {
          for( int i = 0; i < 4; i++ )
            tempbuffer[i] = inputptr[i];
         tempbuffer[i] = NULL;
         sscanf( tempbuffer, "%ld", &direction );
         inputptr = &inputptr[i];
      }

      // read in the speed
```

```
    if( *inputptr != NULL && *inputptr != 10 ) {
        for( i = 0; i < 7; i++ )
          tempbuffer[i] = inputptr[i];
       tempbuffer[i] = NULL;
         sscanf( tempbuffer, "%f", &speed );
       inputptr = &inputptr[i];
    }

    //read in the gusts
    if( *inputptr != NULL && *inputptr != 10 ) {
       for( i = 0; i < 7; i++ )
          tempbuffer[i] = inputptr[i];
       tempbuffer[i] = NULL;
       sscanf( tempbuffer, "%f", &gust );
       inputptr = &inputptr[i];
    }

    // read in the DDEV
    if( *inputptr != NULL && *inputptr != 10 ) {
       for( i = 0; i < 6; i++ )
          tempbuffer[i] = inputptr[i];
       tempbuffer[i] = NULL;
       sscanf( tempbuffer, "%f", &ddev );
       inputptr = &inputptr[i];
    }

    // read in the temp
    if( *inputptr != NULL && *inputptr != 10 ) {
       for( i = 0; i < 7; i++ )
          tempbuffer[i] = inputptr[i];
       tempbuffer[i] = NULL;
       sscanf( tempbuffer, "%f", &temp );
       inputptr = &inputptr[i];
    }
    // read in the dew point temp
    if( *inputptr != NULL && *inputptr != 10 ) {
       for( i = 0; i < 7; i++ )
          tempbuffer[i] = inputptr[i];
       tempbuffer[i] = NULL;
       sscanf( tempbuffer, "%f", &td );
       inputptr = &inputptr[i];
    }
    // read in the rh
    if( *inputptr != NULL && *inputptr != 10 ) {
       for( i = 0; i < 6; i++ )
          tempbuffer[i] = inputptr[i];
       tempbuffer[i] = NULL;
       sscanf( tempbuffer, "%f", &rh );
    }

// find out the location in the obs data structure to put the data
    idindex = getSensorIndex( id );
    heightindex = getHeightIndex( height );
      if( idindex == -1 || heightindex == -1 )
        idindex = 0;// cout << "index error" << endl;
```

```
            //  the id or height is not valid

         else
              if( direction != -1 && speed != -1 ) {
                  // make sure wind data present
                  if( direction >= 180 )
                     direction -= 180;
                  else
                     direction += 180;
                  currObs->data[idindex][heightindex].setWindV(
direction, speed );
               }
        // save this information in order to make next link in list
          lastTime = time;
        }

       // get user input to determine if more data files are ready

    }
    fclose( indatafile );
   return t;   // return the number of time steps
}


//  function:  getsensorindex
//  This function uses the passed id to locate the index of that
//     sensor in the 3-D wind field.  This information is based upon the
//     data read in from the sensor file
//
int ObsWindField::getSensorIndex( long id ) {

   int i;

   for( i = 0; i < NUM_SENSORS; i++ )
      if( sensorInfo[i].id == id )
        break;
   if( i == NUM_SENSORS )
      i = -1;
   return i;
}
//  function:  getheightindex
//  This function uses the passed height to locate the index of that
//     sensor in the 3-D wind field.  This information is based upon the
//     global array located in the header file
//

int ObsWindField::getHeightIndex( long height ) {

   int i;

   for( i = 0; i < NUM_HEIGHTS; i++ )
     if( heightIndex[i] == height )
        break;
   if( i == NUM_HEIGHTS )
      i = -1;
```

```
   return i;
}

//  function: display_input
//  This function loops through the 3-D input array of the observed
//     values and displays the values of the wind field input
//
void ObsWindField::displayInput() {
    float u,v;
    obsListType *temp;

    temp = obsListHead;
    while( temp != NULL ) {
//      cout << "time step" << endl;
        temp = temp->next;
    }
//  getchar();


   for( int i = 0; i < NUM_SENSORS; i++ ) {
      for( int j = 0; j < NUM_HEIGHTS; j++ ) {
           currObs->data[i][j].getWind( &u, &v );
           if( u == -999.0 )
               printf( "   n/a " );
               // cout << "   n/a ";      // no wind data read in
           else
               printf( "% 2.2f ",u);
               // printf( "% 2.2f/% 2.2f ",u,v );
               //;cout << u << "/" << v << " ";
       }
      printf("\n");
    }

    getchar();
}

//  function:  getSensorData
//  This function reads in the specifications about the wind sensors.
//     This data is kept in the file sensor.dat.  Each line contains the
//     id number, lat,lat minutes, lat secondds, lon, lon minutes, lon
// seconds, height
void ObsWindField::getSensorData() {
    float lat,latm,lats,lon,lonm,lons,height;
   int id,i=0;

   FILE *sensorFile;

      sensorFile = fopen( "sensor.dat", "r" );

      // make sure the file will open
       if( !sensorFile ) {
         cout << "Sensor File could not be opened" << endl;
         exit(1);      // exit loop and program
      }
```

```
      // read in the data from the file
       fscanf( sensorFile, "%d %f %f %f %f %f %f %f",
            &id, &lat, &latm, &lats, &lon, &lonm, &lons, &height );

       // this loop reads in data and loads it into the sensor
       //    data structure until the file is empty
      do {
           sensorInfo[i].id = id;
           sensorInfo[i].latitude = (float) (lat + latm/60.0 +
lats/6000.0);
           sensorInfo[i].longitude = (float) (lon + lonm/60.0 +
lons/6000.0);
           sensorInfo[i].height = height;
           i++;

           // read the data from the file
           fscanf( sensorFile, "%d %f %f %f %f %f %f %f",
                &id, &lat, &latm, &lats, &lon, &lonm, &lons, &height );

       } while( !feof( sensorFile ) && i < NUM_SENSORS );
}


// function: displaySensorInfo
// This function loops through the sensor data structure and prints the
data to the screen

void ObsWindField::displaySensorInfo() {
    for( int i = 0; i<NUM_SENSORS; i++ ) {
        cout << sensorInfo[i].id <<" "<<sensorInfo[i].latitude<<"
"<<sensorInfo[i].longitude<< endl;
    }
    getchar();
}

// Function: collectData
// This function collects the observed data based upon the random number
file
void ObsWindField::collectData() {
    obsListType *obsListPtr;
    obsDataType *currObsPtr;
    FILE * randomNumFile;
    float obsu, obsv;
    int index[NUM_HEIGHTS][NUM_SENSORS];
   int numSensors[NUM_HEIGHTS],t = 0;
    float meanU=0, totObsU=0, meanV=0, totObsV=0, tempU = 0, tempV = 0;
    cout << "in collect data" << endl; // getchar();
    // point to the data in the first time slot
    obsListPtr = obsListHead;
    currObsPtr = obsListPtr->observation;
    // read the random number file
    randomNumFile = fopen( "random.dat", "r" );
    for( int i = 0; i < MAXTIMES; i++ ) {
        fscanf( randomNumFile, "%d ", &idList[i] );
    }
```

```
    fclose( randomNumFile );

    // loop through the time steps
    while( obsListPtr != NULL ) {

    // get the number of sensors per level
    //  outer loop steps through the vertical
    //  the inner loop counts the num of sensors with valid observations
        for( int z = 0; z < NUM_HEIGHTS; z++ ) {

   // initialize the counter for the current levels num of sensors
           numSensors[z] = 0;

// first determine the number of sensors and thier indexes on this level
//   this is the actual number of sensors with accepted observations
           for( int n = 0; n < NUM_SENSORS; n++ ) {
               currObsPtr->data[n][z].getWind( &obsu, &obsv );
               if( obsu != -999.0 && obsv != -999.0 )
                   index[z][numSensors[z]++] = n;
           }
//    only deal with height of maximum observations -- z = 2  - for now
//    save the observation
//    set the observation to -999
           if( z == 2 ) {

            // initialize for each time step
              meanU = meanV = 0;
              totObsV = totObsU = 0;
            tempU = tempV = 0;

              //    calculate the mean value
              for( int n = 0; n < numSensors[z]; n++ ) {

               // get the observation
                  currObsPtr->data[index[z][n]][z].getWind(&obsu,&obsv);

                   // add the new to the running total
                   totObsU += obsu;
               totObsV += obsv;
              }
              meanU = totObsU/numSensors[z];
              meanV = totObsV/numSensors[z];
            mean[t].setWind( meanU, meanV );

            totObsU = totObsV = 0;
            // now calculate the variance and the std
              for( n = 0; n < numSensors[z]; n++ ) {

               // get the observation
              currObsPtr->data[index[z][n]][z].getWind(&obsu,&obsv );

                   // add the new to the running total
                   totObsU += (float) pow( (double)(obsu-meanU),2.0);
                   totObsV += (float) pow( (double)(obsv-meanV),2.0);
              }
```

```
                tempU = totObsU/(numSensors[z]-1);
                tempV = totObsV/(numSensors[z]-1);

                std[t].setWind((float) sqrt((double) tempU), (float)
sqrt((double) tempV) );

                for( n = 0; n < numSensors[z]; n++ ) {

                 // get the observation
                currObsPtr->data[index[z][n]][z].getWind( &obsu, &obsv );

                 // see if the sensor id is the selected one
                    if( n == idList[t]%numSensors[z] ) {
                        idList[t] = sensorInfo[index[z][n]].id;
                        actual[t].setWind( obsu, obsv );
                        currObsPtr->data[index[z][n]][z].setWind( -999.0,
-999.0 );
                    }
                }

            }
          }
           obsListPtr = obsListPtr->next;
          if( obsListPtr != NULL )
               currObsPtr = obsListPtr->observation;
          t++;
      }   // end the outer time loop

/*  for( int w = 0; w < t; w++ ) {
        mean[w].getWind( &obsu, &obsv );
       cout << "mean " << obsu << " " << obsv << endl;
    }
    for(  w = 0; w < t; w++ ) {
        std[w].getWind( &obsu, &obsv );
       cout << "std " << obsu << " " << obsv << endl;
   }  */
}

// function:  test
// This function allow selection of a specific sensor id and height
//    and then displays that wind data to the screen
void ObsWindField::test() {
    char another = 'y';
    int height, id;
    float outu,outv;

        cout << "another? ";
        another = getchar();getchar();  // second one eats the ret char

   while( another != 'n' && another != 'N' ) {
        cout << "ID ";
        cin >> id;
        cout << "height";
        cin >> height;
```

```
        currObs-
>data[getSensorIndex(id)][getHeightIndex(height)].getWind( &outu, &outv
);
        cout << outu << "/" << outv << endl;

        cout << "another? ";
        another = getchar();getchar();  // second one eats the ret char
    }
}

//  function: Barnes
//     This is the constructor function for the Barnes class.  It takes
//     care of allocating the memory for the wind field as well as the
//     correction arrays Two analysis arrays are required for the Barnes
//     system

Barnes::Barnes() {
    // allocate memory for the data structures
    // these structs must be dynamically allocated due to stack limits
    corrFactor = (obsDataType *) malloc(sizeof(obsDataType));
    anal1 = (Field3DType *) malloc(sizeof(Field3DType));
    anal2 = (Field3DType *) malloc(sizeof(Field3DType));
    sensor = (Int2DType *) malloc(sizeof(Int2DType));

    // make sure the memory is present to run the program
    if( corrFactor == NULL || anal1 == NULL || anal2 == NULL || sensor
== NULL) {
          cout << "Insufficient memory to run" << endl;
            exit(1);
    }

    // intialize the correction fields to 0
    for( int i = 0; i < NUM_SENSORS; i++ ) {
       for( int j = 0; j < NUM_HEIGHTS; j++ ) {
            corrFactor->data[i][j].setWind( 0.0, 0.0 );
       }
    }
}

//  function: ~Barnes
//  This function is the destructor for the Barnes class
//  The function checks to make sure the pointers are still pointing to
//  allocated memory.  The destructor for the obs data does the same

Barnes::~Barnes() {

    if( corrFactor ) {
        free( corrFactor );
        malloc(sizeof(obsDataType));
        corrFactor = NULL;
    }
    if( anal1 ) {
        free( anal1 );
        malloc(sizeof(Field3DType));
        anal1 = NULL;
```

```
    }
    if( anal2 ) {
        free( anal2 );
        malloc(sizeof(Field3DType));
        anal2 = NULL;
    }
    if( sensor ) {
        free( sensor );
        malloc(sizeof(Int2DType));
        sensor = NULL;
    }
    // call the destructor for the observation data
    obsWind.~ObsWindField();
}

// function: doBarnes
// This function is the main driving function for the Barnes anaylsis.
// It calls the other functions required to read in the sensor
// information as well as the observed data
void Barnes::doBarnes() {
    Field3DType *tempAnalPtr;          // used for array swapping
    float obsu, obsv, anallru,anallrv;
    float corru, corrv, analulu, analulv, analuru,
analurv,analllu,analllv;
    int iw, js;   // the array index(s) to the west & south of obs point
    obsListType * obsStepPtr;
   int n=0;

   // read in the sensor data (lat/lon) information from the sensor file
    obsWind.getSensorData();

   // display the read in data --  debugging tool
   //  obsWind.displaySensorInfo();

   // get the observed wind data from the data file(s)
    numOfTimeSteps = obsWind.getWindData();


   // get the actual data values
    if( dataCollection )
      obsWind.collectData();

   // interigate the observed winds for accuracy
    // obsWind.test();

   //  display the observed wind field
    // obsWind.displayInput();

    // loop through the time steps
    obsStepPtr = obsWind.obsListHead;
    obsWind.currObs = obsStepPtr->observation;
    output2dAnalFieldHdr( "pass1" );

    while( obsStepPtr != NULL ) {
```

```
        cout << "Barnes Time step " << n << endl;
//  outer loop steps through the vertical
//  the inner loop counts the number of sensors with valid observations
        for( int z = 0; z < NUM_HEIGHTS; z++ ) {

        // initialize the counter for the current levels num of sensors
            numOfSensors[z] = 0;

// first determine the number of sensors and thier indexes on this level
//   this is the actual number of sensors with accepted observations
            for( int n = 0; n < NUM_SENSORS; n++ ) {
            obsWind.currObs->data[n][z].getWind( &obsu, &obsv );
                if( obsu != -999.0 && obsv != -999.0 )
                    sensor->index[z][numOfSensors[z]++] = n;
            }
    }

    // call the barnes subroutine the first time
        barnesSub( (float) K41, XRANGE, YRANGE, NUM_HEIGHTS , 1 );

        // swap pointers to the anal arrays
        tempAnalPtr = anal1;        // save pointer to the first guess
        anal1 = anal2;              // anal1 now contains nothing
        anal2 = tempAnalPtr;        // anal2 now contains the first guess

        // calculate the correction factors
    // outer loop steps through the heights
        for( z = 0; z < NUM_HEIGHTS; z++ ) {
        // inner loop steps throught the sensors
            for( int n = 0; n < numOfSensors[z]; n++ ) {
                iw = ( MAX_LON - obsWind.sensorInfo[sensor-
>index[z][n]].longitude )/LON_STEP;
                js = ( MAX_LAT - obsWind.sensorInfo[sensor-
>index[z][n]].latitude )/LAT_STEP;
                obsWind.currObs->data[sensor-
>index[z][n]][z].getWind(&obsu,&obsv);
                anal2->data[iw][js][z].getWind(&analulu,&analulv);
                anal2->data[iw+1][js][z].getWind(&analuru,&analurv);
                anal2->data[iw][js+1][z].getWind(&analllu,&analllv);
            anal2->data[iw+1][js+1][z].getWind(&anallru,&anallrv);
                corru = obsu-(analulu+analuru+analllu+anallru)/4;
                corrv = obsv-(analulv+analurv+analllv+anallrv)/4;
                corrFactor->data[sensor-
>index[z][n]][z].setWind(corru,corrv);
        }
    }

        // call the barnes subroutine for the second time
        //  this provides for a closer adjustment
        barnesSub( (float) K42, XRANGE, YRANGE, NUM_HEIGHTS, 2 );

        // now add the two together to get the final estimate
        for( z = 0; z < NUM_HEIGHTS; z++ ) {
        for( int i = 0; i < XRANGE; i++ ) {
            for( int j = 0; j < YRANGE; j++ ) {
```

```
                    anal2->data[i][j][z].getWind(&obsu,&obsv);
                    anal1->data[i][j][z].getWind(&corru,&corrv);
                    anal1-
>data[i][j][z].setWind((obsu+corru),(obsv+corrv));
                    // anal1 now has the final guess
                }
            }
            }

            if( dataCollection )
                getAnalData( n, XRANGE, YRANGE );


            // insert code to print the current time slot data to a data
file
            cout << "Print Barnes Time step " << n++ << endl;

            output2dAnalField( "pass1");

           // step through the linked list
            obsStepPtr = obsStepPtr->next;
           if( obsStepPtr != NULL )
               obsWind.currObs = obsStepPtr->observation;
       } // end wile loop for time

}

// function: barnesSub
// This function performs the meat of the barnes routine
void Barnes::barnesSub( float k4, int xrange, int yrange, int zrange,
int pass) {
  float beta[NUM_SENSORS], w[NUM_SENSORS];
  float wsum = 0.0, glat,glon,tempu,tempv,obsu,obsv;

  // outer loop steps through the vertical
  for( int z = 0; z < zrange; z++ ) {

      // now perform the analysis
      // loop through the 2-D field
      for( int j = 0; j < yrange; j++ ) {
          glat = MIN_LAT+LAT_STEP*j;
          for( int i = 0; i < xrange; i++ ){
              glon = MIN_LON+LON_STEP*i;
              wsum = 0.0;

            // loop through the sensors to determine the raw weights
              for( int n = 0; n < numOfSensors[z]; n++ ) {
                  w[sensor->index[z][n]] = exp( -1.0*
                      dist( obsWind.sensorInfo[sensor-
>index[z][n]].latitude,
                      obsWind.sensorInfo[sensor->index[z][n]].longitude,
glat, glon)/
                                          (k4*1.2/numOfSensors[z]));
                  wsum = wsum + w[sensor->index[z][n]];
              }
```

```
            // loop through the sensors to calculate the final weight
              for( n = 0; n < numOfSensors[z]; n++ ) {
                 beta[sensor->index[z][n]] = w[sensor->index[z][n]]/wsum;
              }

            // initialize the analysed field
              tempu = tempv = 0.0;

            // now step through the sensors to calcualate the estimate
              for( n = 0; n < numOfSensors[z]; n++ ) {
                  if( pass == 1 )
                      obsWind.currObs->data[sensor-
>index[z][n]][z].getWind(&obsu,&obsv);
                  else
                      corrFactor->data[sensor-
>index[z][n]][z].getWind(&obsu,&obsv);
                  tempu = tempu + beta[sensor->index[z][n]]*obsu;
                  tempv = tempv + beta[sensor->index[z][n]]*obsv;
            }
              anal1->data[i][j][z].setWind( tempu, tempv );
        }
      }
   }
}

// function: dist
//  This function accepts in the lat lon pairs and calculates the
distance
//  between them in degrees latitude squared
float Barnes::dist( float lat1, float lon1, float lat2, float lon2 ) {
//   cout << lat1 << " " << lon1 << " " << lat2 << " " << lon2 << endl;
    return (pow((lat2-
lat1),2.0)+pow(cos((double)(lat1+lat2)*PI/(2.0*180.0)),2.0)*
            pow((lon2-lon1),2.0));
}

// Function: getAnalData
// This function collects the data on the
void Barnes::getAnalData( int t, int xrange, int yrange ) {
    float meanU, meanV, stdU = 0, stdV = 0, tempu, tempv;
    float totAnalU, totAnalV;
   FILE * datafile;

   int z = 2;   // only look at the second level -- for now

  // cout << " in get anal data "<< endl;

   totAnalU = totAnalV = 0;
   //    calculate the mean value
   for( int i = 0; i < xrange; i++ ) {
       for( int j = 0; j < yrange; j++ ) {

        // get the analized values
         anal1->data[i][j][z].getWind( &tempu, &tempv );
```

```
        // add the new to the running total
        totAnalU += tempu;
        totAnalV += tempv;
      }
    }
    meanU = totAnalU/(xrange*yrange);
    meanV = totAnalV/(xrange*yrange);

   // cout << meanU << " " << meanV << endl;

    totAnalU = totAnalV = 0;
    // now calculate the variance and the std
    for( i = 0; i < xrange; i++ ) {
        for( int j = 0; j < yrange; j++ ) {
        // get the observation
            anal1->data[i][j][z].getWind( &tempu, &tempv );

            // add the new to the running total
                   totAnalU += (float) pow( (double)(tempu-meanU),2.0);
                   totAnalV += (float) pow( (double)(tempv-meanV),2.0);
       }
     }
     stdU = (float) sqrt( (double) (totAnalU/(xrange*yrange)));
     stdV = (float) sqrt( (double) (totAnalV/(xrange*yrange)));
//   cout << stdU << " " << stdV << endl;

float glat, glon, tempLat = MIN_LAT, tempLon = MIN_LON;
float leftU, leftV, rightU, rightV, estU, estV, temp2u, temp2v;
int p, q;

   // find the analysed value
   // first get the lat/lon of the withdeld observation
     for( int n = 0; n < NUM_SENSORS; n++ ) {

     // get the observation
         if( idList[t] == obsWind.sensorInfo[n].id ) {
            glat =  obsWind.sensorInfo[n].latitude;
            glon =  obsWind.sensorInfo[n].longitude;
          break;
     }
     }

     // now get the location of the four surrounding lat/lon pairs
     // find the latitude index
     for( p = 0; p < XRANGE; p++ ) {
         if( (tempLat + LAT_STEP) > glat )
             break;          // found lower
        tempLat += LAT_STEP;
     }
     // find the longitude index
     for( q = 0; q < YRANGE; q++ ) {
         if( (tempLon + LON_STEP) > glon )
             break;          // found left
        tempLon += LON_STEP;
     }
```

```
// cout << glat << " " << tempLat << " "  << glon << " " << tempLon <<
endl;
    //  now perform the linear interpolation to estimate the estimate

    // get the wind values on the west side
    anal1->data[p][q+1][z].getWind( &tempu, &tempv );
    anal1->data[p][q][z].getWind( &temp2u, &temp2v );

    // interpolate the U and V values
   leftU = temp2u + ((glat - tempLat)/(float)LAT_STEP)*(tempu - temp2u);
    leftV = temp2v + ((glat - tempLat)/(float)LAT_STEP)*(tempv-temp2v);

    // get the wind values on the east side
    anal1->data[p+1][q+1][z].getWind( &tempu, &tempv );
    anal1->data[p+1][q][z].getWind( &temp2u, &temp2v );

    // interpolate the U and V values
   rightU = temp2u + ((glat - tempLat)/(float)LAT_STEP)*(tempu- temp2u);
    rightV = temp2v + ((glat-tempLat)/(float)LAT_STEP)*(tempv- temp2v);

    // now get the final estimate
   estU = leftU + ((glon - tempLon)/(float)LON_STEP)*(rightU - leftU);
    estV = leftV + ((glon - tempLon)/(float)LON_STEP)*(rightV - leftV);

              //*/
   datafile = fopen( "anal.dat", "a" );
   actual[t].getWind( &tempu, &tempv );
    fprintf( datafile, "%d %f %f ", idList[t], tempu, tempv );
    mean[t].getWind( &tempu, &tempv );
    fprintf( datafile, "%f %f ", tempu, tempv );
    std[t].getWind( &tempu, &tempv );
    fprintf( datafile, "%f %f ", tempu, tempv );
    fprintf( datafile, "%f %f ", estU, estV );
    fprintf( datafile, "%f %f ", meanU, meanV );
    fprintf( datafile, "%f %f\n", stdU, stdV );
    fclose( datafile );

}

// function: displayAnalField
// This function gets the user input for the level and then
// displays the analysed winds at the level specified
void Barnes::displayAnalField() {
    float tempu, tempv;
   int level;

    for( int z = 0; z < NUM_HEIGHTS; z++ )
        cout << numOfSensors[z] << " " ;

    cout << "enter level? ";
    cin >> level;

    while( level != 99 ) {
        for( int j = 0; j < YRANGE; j++ ) {
```

```
            for( int i = 0; i < XRANGE; i++ ) {
                anal1->data[i][j][level].getWind( &tempu, &tempv );
                printf( "% 2.2f ",tempu );
                //printf( "% 2.2f/% 2.2f ",tempu,tempv );
            }
             printf( "\n");
    }
        cout << "enter level? ";
        cin >> level;
    }
}

// function: output2dAnalFieldHdr
// This function prints the required information concerning number of
time steps
// size of the output data fields, lat/lon data, lat/lon step size, ...

int Barnes::output2dAnalFieldHdr( char * filename ) {
    FILE * outdatafile;
    obsListType * obsStepPtr;


    // open the file -- overwrite any existing file

   outdatafile = fopen( filename, "w" );

   // check to make sure the file opened
   if( !outdatafile ) {
      cout << "File could not be opened" << endl;
      exit(1);      // exit loop and program
    }
   cout << "Printing Header info to" << filename << endl;
    // now print the appropriate data
   // number of time steps -- 50 max
    fprintf( outdatafile, "%d\n", numOfTimeSteps );
   // number of variables
    fprintf( outdatafile, "2\n");
   // the data array dimmensions
    fprintf( outdatafile, "%d %d %d\n", XRANGE, YRANGE, NUM_HEIGHTS);

    // variable names
    fprintf( outdatafile, "U\nV\n" );
    // time stamps for each time step
    // must loop through and print
    obsStepPtr = obsWind.obsListHead;
    while( obsStepPtr ) {
        fprintf( outdatafile, "%ld ", obsStepPtr->observation->time );
       obsStepPtr = obsStepPtr->next;
    }
    fprintf( outdatafile, "\n" );
    // date stamp for each data set
    // must loop through and print
    obsStepPtr = obsWind.obsListHead;
    while( obsStepPtr ) {
        fprintf( outdatafile,"%ld ", obsStepPtr->observation->date );
```

```
      obsStepPtr = obsStepPtr->next;
   }
   // northern most latitude
   fprintf( outdatafile, "\n%f\n", MAX_LAT);
   // the size of the incriment for latitude steps
   fprintf( outdatafile, "%f\n", LAT_STEP );
   // western most longitude
   fprintf( outdatafile, "%f\n",MIN_LON-360.0 );
   // the size of the incriment for latitude steps
   fprintf( outdatafile, "%f\n", LON_STEP);
   //height of the lowest data AGL -- 12 feet
   fprintf( outdatafile, "0.0036576\n");
   fprintf( outdatafile, "%f\n", ALT_STEP*0.0003048);
   fclose( outdatafile );
   return 1;
}

// function: output2dAnalField
// This function prints the actual data to the first pass file.  This
//  data has not
// been interpolated vertically to the equally spaced vertical grid

int Barnes::output2dAnalField( char * filename) {
   FILE * outdatafile;
   float outu, outv;

   // begin the outer loop -- allows multiple files to be read in
   // open the file to append to it

   outdatafile = fopen( filename, "a" );

   // check to make sure the file opened
   if( !outdatafile ) {
      cout << "File could not be opened" << endl;
      exit(1);      // exit loop and program
    }

    // first print the u value for every height
   for( int z = 0; z < NUM_HEIGHTS; z++ ) {
       for( int j = 0; j < YRANGE; j++ ) {
           for( int i = 0; i < XRANGE; i++ ) {
              anal1->data[i][j][z].getWind( &outu, &outv );
           fprintf( outdatafile, "% f ", outu );
           }
         fprintf( outdatafile, "\n" );
       }
   }

   // then print out the v value at each height
    for( z = 0; z < NUM_HEIGHTS; z++ ) {
       for( int j = 0; j < YRANGE; j++ ) {
           for( int i = 0; i < XRANGE; i++ ) {
              anal1->data[i][j][z].getWind( &outu, &outv );
           fprintf( outdatafile, "% f ",outv );
           }
```

```
            fprintf( outdatafile, "\n" );
         }
      }
      fclose( outdatafile );

      return( 1 );
}

// Function: doVerticalInterpolation
// This function reads the data file pass1 and performs the vertical
interpolation
//    using a cubic spline routine
void doVerticalInterpolation( void ) {
      float *g;
      float *final;
      FILE *f;
      FILE *o;
      int it, iv, ir, ic, il;

/** STEP 1:  The following variables must be initialized in STEP 2.  See
 ** the README file section describing the 'v5dCreateSimple' call for
 ** more information.
 **/
      int NumTimes;                        /* number of time steps */
      int NumVars;                         /* number of variables */
      int Nr, Nc, Nl;                      /* size of 3-D grids */
      char VarName[MAXVARS][10];          /* names of variables */
      long int TimeStamp[MAXTIMES];       /* real times for each time step */
      long int DateStamp[MAXTIMES];       /* real dates for each time step */
      float NorthLat;                      /* latitude of north bound of box */
      float LatInc;                        /* spacing between rows in degrees */
      float WestLon;                       /* longitude of west bound of box */
      float LonInc;                        /* spacing between columns in degs */
      float BottomHgt;                     /* height of bottom of box in km */
      float HgtInc;                     /* spacing between grid levels in km */

float *t, *y, *h, *b, *u, *v, *z; // pointers to work arrays used to
// perform the cubic spline

      cout << "Performing Vertical Interpolation" << endl;

      // open the first pass file
      // read in all of the header information
      f = fopen( "pass1", "r" );
      if (!f) {
         printf("Error: couldn't open %s for reading\n", "pass1" );
         exit(1);
      }
      // read in the header data from the first pass file
      fscanf( f, "%d ", &NumTimes );     /* number of time steps */
      fscanf( f, "%d ", &NumVars );      /* number of variables */
      fscanf( f, "%d %d %d", &Nr, &Nc, &Nl );
      for( int i = 0; i < NumVars; i++ )
         fscanf( f, "%s ", &VarName[i] );     /* names of variables */
      for( i = 0; i < NumTimes; i++ )
```

```
      fscanf( f, "%ld ", &TimeStamp[i] ); /*real times each time step */
   for( i = 0; i < NumTimes; i++ )
      fscanf( f, "%ld ", &DateStamp[i] ); /*real dates each time step */
   fscanf( f, "%f ", &NorthLat );  /* latitude of north bound of box */
   fscanf( f, "%f ", &LatInc );    /* spacing between rows in degrees */
   fscanf( f, "%f ", &WestLon );   /* longitude of west bound of box */
   fscanf( f, "%f ", &LonInc );    /* spacing between columns in degs */
   fscanf( f, "%f ", &BottomHgt ); /* height of bottom of box in km */
   fscanf( f, "%f ", &HgtInc );  /* spacing between grid levels in km */

   /* now open the output file and write the header data to it */

   o = fopen( "pass2", "w" );
   if (!f) {
      printf("Error: couldn't open %s for reading\n", "pass2" );
      exit(1);
   }

   fprintf( o, "%d\n", NumTimes );    /* number of time steps */
   fprintf( o, "%d\n", NumVars );     /* number of variables */
   fprintf( o, "%d %d %d\n", Nr, Nc, Nl );
   for( i = 0; i < NumVars; i++ )
       fprintf( o, "%s\n", VarName[i] );    /* names of variables */
   for( i = 0; i < NumTimes; i++ )
       fprintf( o, "%ld ", TimeStamp[i] );  /* real times for each time
step */
   fprintf( o, "\n" );
   for( i = 0; i < NumTimes; i++ )
       fprintf( o, "%ld ", DateStamp[i] );  /* real dates for each time
step */
   fprintf( o, "\n" );
   fprintf( o, "%f\n", NorthLat ); /* latitude of north bound of box */
   fprintf( o, "%f\n", LatInc );  /* spacing between rows in degrees */
   fprintf( o, "%f\n", WestLon );  /* longitude of west bound of box */
   fprintf( o, "%f\n", LonInc );  /* spacing between columns in degs */
   fprintf( o, "%f\n", BottomHgt ); /* height of bottom of box in km */
   fprintf( o, "%f\n", HgtInc ); /* spacing between grid levels in km */

   /* allocate space for grid data */
   t = (float *) malloc( Nl * sizeof(float) ); // the z data locations
   y = (float *) malloc( Nl * sizeof(float) ); // the data at that
location
   h = (float *) malloc( Nl * sizeof(float) ); // work array
   b = (float *) malloc( Nl * sizeof(float) ); // work array
   u = (float *) malloc( Nl * sizeof(float) ); // work array
   v = (float *) malloc( Nl * sizeof(float) ); // work array
   z = (float *) malloc( Nl * sizeof(float) ); // contains the spline
coefficients
   if (!t || !y || !h || !b || !u || !v || !z ) {
      printf("Error: out of memory\n");
      exit(1);
   }

   // initialize t variable
   for( i = 0; i < Nl; i++ )
```

```
      t[i] = heightIndex[i]; //height index is a global array variable
// alloc the mem for the working arrays used to perform the cubic spline
   g = (float *) malloc( Nr * Nc *  Nl * sizeof(float) );
   if (!g) {
      printf("Error: out of memory\n");
      exit(1);
   }

// alloc the mem for the working arrays used to perform the cubic spline
   final = (float *) malloc( Nr * Nc *  ZRANGE * sizeof(float) );
   if (!final) {
      printf("Error: out of memory\n");
      exit(1);
   }
#define G(ROW, COLUMN, LEVEL) g[(ROW) + ((COLUMN) + (LEVEL) * Nc) * Nr ]
#define Final(ROW, COLUMN, LEVEL) final[(ROW)+((COLUMN)+(LEVEL)*Nc)*Nr]

// cout << "1" << endl; getchar();
   // read in the data from the first pass file

   // loop through the number of time steps
   for (it=0;it<NumTimes;it++) {

   // loop through the two variables (u and v)
      for (iv=0;iv<NumVars;iv++) {

   // read the data values from the pass1 file into the input buffer
            for( il = 0; il < Nl; il++ ) {
                for( ic = 0; ic < Nc; ic++ ) {
                   for( ir = 0; ir < Nr; ir++ )
                       fscanf( f, "%f ", &G( ir, ic, il ) );
             }
            }

            for( ir = 0; ir < Nr; ir++ ) {
                for( ic = 0; ic < Nc; ic++ ) {
                    // load the vertical data into the work array y
                    for( il = 0; il < Nl; il++ )
                        y[il] = G( ir, ic, il );
                    // now call the cubic spline routine
                    cubicSpline( Nl, t, y, h, b, u, v, z );
               // now loop through the desired data locations and call
                    // the interpolation routine
                    // load the results in the final buffer
                 for( il = 0; il < ZRANGE; il ++ ) {
                       Final( ir, ic, il ) = InterpSpline( Nl, t, y, z,

interpHeightIndex[il] );
                       if(il != ZRANGE-1 && outputvert ) {
                          InterpSpline( Nl, t, y,
z,interpHeightIndex[il]+4 );
                       InterpSpline( Nl, t, y,
z,interpHeightIndex[il]+8 );
                             InterpSpline( Nl, t, y,
z,interpHeightIndex[il]+12 );
```

```
                                InterpSpline( Nl, t, y,
z,interpHeightIndex[il]+16 );
                                InterpSpline( Nl, t, y,
z,interpHeightIndex[il]+20 );
                            }
                            else
                            outputvert = 0;
                        }
                    }
                }
              // print the results
                for( il = 0; il < ZRANGE; il++ ) {
                    for( ic = 0; ic < Nc; ic++ ) {
                        for( ir = 0; ir < Nr; ir++ )
                            fprintf( o, "%f ", Final( ir, ic, il ) );
                     fprintf( o, "\n" );
                    }
                }
            }
        }

    fclose(f);
    fclose(o);

}

void cubicSpline( int n, float * t,float * y,float * h,float * b,
                        float * u,float * v,float * z ) {
// cout << "In cubic spline" << endl;

    for( int i = 0; i < n-1; i++ ) {
        h[i] = t[i+1]-t[i];
        b[i] = (y[i+1]-y[i])/h[i];
    }
    u[1] = (float) 2.0*(h[0]+h[1]);
    v[1] = (float) 6.0*(b[1]-b[0]);
    for( i = 2; i < n-1; i++ ) {
       u[i] = 2.0*(h[i]+h[i-1])-(float) pow((double) h[i-1],2.0)/u[i-1];
       v[i] = 6.0*(b[i]-b[i-1])-h[i-1]*v[i-1]/u[i-1];
    }
    z[n-1] = 0.0;
    for( i = n-2; i > 0; i-- ) {
        z[i] = (v[i]-h[i]*z[i+1])/u[i];
    }
   z[0] = 0.0;
}

// Function: InterpSpline
// This function uses the coefficients loaded in the z array to
calculate the
//   interpolated value at the location specified in variable x
float InterpSpline( int n, float * t,float * y,float * z, int x ) {
    float diff, h, b, p;
   int i;
```

```
//   cout << "interp" << endl;
   // first find the appropriate portion of the spline
    for( i = n-2; i > 0; i-- ) {
        diff = x - t[i];
        if( diff > 0.0 )
           break;
    }
    if( !i )
      diff = x - t[i];
    // perform the interpolation
    h = t[i+1]-t[i];
   b = (y[i+1]-y[i])/h-h*(z[i+1]+2.0*z[i])/6.0;
    p = 0.5*z[i]+diff*(z[i+1]-z[i])/(6.0*h);
    p = b+diff*p;

   if( outputvert ) {
        FILE *oo;
        oo = fopen( "vert.dat" , "a" );
        if( !oo ) {
            cout << "Error opening vert.dat file"  ;
        exit(1);
    }
        fprintf( oo, "%d %f\n", x, (y[i]+diff*p));
        fclose( oo );
   }

  // return the results
    return (y[i]+diff*p);
}


//  function: Kriging
//     This is the constructor function for the Kriging class.  It takes
//  care of allocating the memory for the wind field as well as the / //
//  correction arrays. Two analysis arrays are required for the Kriging
// system
Kriging::Kriging() {
// allocate memory for the data structures
// these structures must be dynamically allocated due to local stack
// limitation
    anal1 = (Field3DType *) malloc(sizeof(Field3DType));
    anal2 = (Field3DType *) malloc(sizeof(Field3DType));
    sensor = (Int2DType *) malloc(sizeof(Int2DType));
   // make sure the memory is present to run the program
    if( anal1 == NULL || anal2 == NULL || sensor == NULL) {
         cout << "Insufficient memory to run" << endl;
           exit(1);
    }
}
//  function: ~Kriging
//  This function is the destructor for the Kriging class
//  The function checks to make sure the pointers are still pointing to
//  allocated memory.  The destructor for the observation data does the
same
```

```
Kriging::~Kriging() {
    if( anal1 ) {
        free( anal1 );      //= (Field3DType *)
malloc(sizeof(Field3DType));
       anal1 = NULL;
    }
    if( anal2 ) {
        free( anal2 );      //= (Field3DType *)
malloc(sizeof(Field3DType));
        anal2 = NULL;
    }
    if( sensor ) {
        free( sensor );     //sensor = (Int2DType *)
malloc(sizeof(Int2DType));
        sensor = NULL;
    }
    // call the destructor for the observation data
    obsWind.~ObsWindField();
}

// function: doKriging
// This function is the main driving function for the Kriging anaylsis.
It calls
//    the other functions required to read in the sensor information as
well as the
//    observed data
void Kriging::doKriging() {
    Field3DType *tempAnalPtr;          // used for array swapping
    float obsu, obsv, anallru,anallrv;
    float corru, corrv, analulu, analulv, analuru,
analurv,analllu,analllv;
    int iw, js;    // the array index(s) to the west and south of obs
point
    obsListType * obsStepPtr;
   int n=0;

   // read in the sensor data (lat/lon) information from the sensor file
    obsWind.getSensorData();

   // display the read in data --  debugging tool
   //  obsWind.displaySensorInfo();

    // get the observed wind data from the data file(s)
    numOfTimeSteps = obsWind.getWindData();
    // interigate the observed winds for accuracy
    // obsWind.test();

    // get the actual data values
    if( dataCollection )
      obsWind.collectData();

   //  display the observed wind field
    // obsWind.displayInput();

    // loop through the time steps
```

```
    obsStepPtr = obsWind.obsListHead;
    obsWind.currObs = obsStepPtr->observation;
    output2dAnalFieldHdr( "pass1k" );

   // outer loop steps through the time steps
    while( obsStepPtr != NULL ) {

        cout << "Kriging Time step " << n << endl;
    //  this loop steps through the vertical
    //  the inner loop counts the number of sensors with valid obs
        for( int z = 0; z < NUM_HEIGHTS; z++ ) {

        // initialize the counter for the current levels num of sensors
            numOfSensors[z] = 0;

    // first determine the num of sensors and thier indexes onthis level
    //   this is the actual number of sensors with accepted observations
            for( int n = 0; n < NUM_SENSORS; n++ ) {
            obsWind.currObs->data[n][z].getWind( &obsu, &obsv );
                if( obsu != -999.0 && obsv != -999.0 )
                    sensor->index[z][numOfSensors[z]++] = n;
            }
    }

        // call the Kriging subroutine the first time
        KrigingSub( XRANGE, YRANGE, NUM_HEIGHTS );

        if( dataCollection )
            getAnalData( n, XRANGE, YRANGE );

    // insert code to print the current time slot data to a data file
        cout << "Print Kriging Time step " << n++ << endl;

        output2dAnalField( "pass1k");

        // step through the linked list
        obsStepPtr = obsStepPtr->next;
       if( obsStepPtr != NULL )
            obsWind.currObs = obsStepPtr->observation;
    } // end wile loop for time

}

// Function: getAnalData
// This function collects the data on the
void Kriging::getAnalData( int t, int xrange, int yrange ) {
    float meanU, meanV, stdU = 0, stdV = 0, tempu, tempv;
    float totAnalU, totAnalV;
   FILE * datafile;

   int z = 2;   // only look at the second level -- for now

  // cout << " in get anal data "<< endl;

   totAnalU = totAnalV = 0;
```

```
    //    calculate the mean value
    for( int i = 0; i < xrange; i++ ) {
        for( int j = 0; j < yrange; j++ ) {

         // get the analized values
          anal1->data[i][j][z].getWind( &tempu, &tempv );
         // add the new to the running total
         totAnalU += tempu;
         totAnalV += tempv;
      }
    }
    meanU = totAnalU/(xrange*yrange);
    meanV = totAnalV/(xrange*yrange);

   // cout << meanU << " " << meanV << endl;

    totAnalU = totAnalV = 0;
    // now calculate the variance and the std
    for( i = 0; i < xrange; i++ ) {
        for( int j = 0; j < yrange; j++ ) {
        // get the observation
            anal1->data[i][j][z].getWind( &tempu, &tempv );

            // add the new to the running total
                   totAnalU += (float) pow( (double)(tempu-meanU),2.0);
                   totAnalV += (float) pow( (double)(tempv-meanV),2.0);
       }
     }
     stdU = (float) sqrt( (double) (totAnalU/(xrange*yrange)));
     stdV = (float) sqrt( (double) (totAnalV/(xrange*yrange)));

float glat, glon, tempLat = MIN_LAT, tempLon = MIN_LON;
float leftU, leftV, rightU, rightV, estU, estV, temp2u, temp2v;
int p, q;

   // find the analysed value
   // first get the lat/lon of the withdeld observation
    for( int n = 0; n < NUM_SENSORS; n++ ) {

    // get the observation
        if( idList[t] == obsWind.sensorInfo[n].id ) {
            glat =  obsWind.sensorInfo[n].latitude;
            glon =  obsWind.sensorInfo[n].longitude;
          break;
    }
    }

    // now get the location of the four surrounding lat/lon pairs
    // find the latitude index
    for( p = 0; p < XRANGE; p++ ) {
        if( (tempLat + LAT_STEP) > glat )
            break;          // found lower
       tempLat += LAT_STEP;
    }
     // find the longitude index
```

```
    for( q = 0; q < YRANGE; q++ ) {
        if( (tempLon + LON_STEP) > glon )
            break;          // found left
       tempLon += LON_STEP;
    }

    //  now perform the linear interpolation to estimate the estimate

    // get the wind values on the west side
    anal1->data[p][q+1][z].getWind( &tempu, &tempv );
    anal1->data[p][q][z].getWind( &temp2u, &temp2v );

    // interpolate the U and V values
   leftU = temp2u + ((glat - tempLat)/(float)LAT_STEP)*(tempu - temp2u);
    leftV = temp2v + ((glat - tempLat)/(float)LAT_STEP)*(tempv -temp2v);

    // get the wind values on the east side
    anal1->data[p+1][q+1][z].getWind( &tempu, &tempv );
    anal1->data[p+1][q][z].getWind( &temp2u, &temp2v );

    // interpolate the U and V values
   rightU = temp2u + ((glat - tempLat)/(float)LAT_STEP)*(tempu- temp2u);
    rightV = temp2v + ((glat - tempLat)/(float)LAT_STEP)*(tempv-temp2v);

    // now get the final estimate
   estU = leftU + ((glon - tempLon)/(float)LON_STEP)*(rightU - leftU);
    estV = leftV + ((glon - tempLon)/(float)LON_STEP)*(rightV - leftV);

   datafile = fopen( "anal.dat", "a" );
   actual[t].getWind( &tempu, &tempv );
    fprintf( datafile, "%d %f %f ", idList[t], tempu, tempv );
    mean[t].getWind( &tempu, &tempv );
    fprintf( datafile, "%f %f ", tempu, tempv );
    std[t].getWind( &tempu, &tempv );
    fprintf( datafile, "%f %f ", tempu, tempv );
    fprintf( datafile, "%f %f ", estU, estV );
    fprintf( datafile, "%f %f ", meanU, meanV );
    fprintf( datafile, "%f %f\n", stdU, stdV );
    fclose( datafile );

}
// these variobles are global here
float semiVario[NUM_SENSORS+1][NUM_SENSORS+1];
float semiVarioInv[NUM_SENSORS+1][NUM_SENSORS+1];

// function: KrigingSub
// This function performs the meat of the Kriging routine
void Kriging::KrigingSub( int xrange, int yrange, int zrange) {
    float glat,glon,tempu,tempv,obsu,obsv, distance, covar;
    int x, y, matrixSize;
//float semiVario[NUM_SENSORS+1][NUM_SENSORS+1];
//float semiVarioInv[1][1];
    float w[NUM_SENSORS+1];
    float d[NUM_SENSORS+1];
    int pivot;
```

```
    float big, dummy;
    float sum;

  // outer loop steps through the vertical
    for( int z = 0; z < zrange; z++ ) {

        // now perform the analysis
        // loop through the 2-D field

// if there are no sensors at the level skip it.
        if( numOfSensors[z] == 0 )
          continue;
        matrixSize = numOfSensors[z]+1;

        // load the coVariance matrix for this level
        for( int n = 0; n < numOfSensors[z]; n++ ) {
            for( int l = n; l < numOfSensors[z]; l++ ) {
                obsWind.currObs->data[sensor-
>index[z][n]][z].getWind(&obsu,&obsv);
                obsWind.currObs->data[sensor-
>index[z][l]][z].getWind(&tempu,&tempv);

             // get the distance between the sensors
                distance = dist( obsWind.sensorInfo[sensor-
>index[z][n]].latitude,
                      obsWind.sensorInfo[sensor->index[z][n]].longitude,
                      obsWind.sensorInfo[sensor->index[z][l]].latitude,
                      obsWind.sensorInfo[sensor->index[z][l]].longitude );

                covar = coVariance( distance );
                semiVario[n][l] = covar;
                semiVario[l][n] = covar;
            }
        }

        // finish initializing the last row/column of the matrix
        for( n = 0; n < numOfSensors[z]; n++ ) {
            semiVario[n][numOfSensors[z]] = 1.0;
            semiVario[numOfSensors[z]][n] = 1.0;
        }
        semiVario[numOfSensors[z]][numOfSensors[z]]= 0.0;

       if( display ) {
        // invert the matrix
            printf( "\n" );
            for(int q = 0; q<matrixSize; q++ ) {
                for(int p = 0; p<matrixSize; p++ ) {
               printf( "%2.6f ", semiVario[p][q] );
                }
                printf( "\n");
            }
       }

       // the inversed matrix starts out as the identity matrix
        for( int i = 0; i < matrixSize; i++ ) {
```

```
            for( int k=0; k< matrixSize; k++ ) {
                if( k == i )
                    semiVarioInv[i][k] = 1.0;
                else
                    semiVarioInv[i][k] = 0.0;
            }
        }
    // start the matrix inversion routine

    for( int k = 0; k < matrixSize; k++ ) {

       // perform partial pivoting
           pivot = k;
           big = (float) fabs( semiVario[k][k]);

           for( int ii = k +1; ii < matrixSize; ii++ ) {
               dummy = (float) fabs( semiVario[ii][k] );
               if( dummy > big ) {
                   big = dummy;
                pivot = ii;
           }
           }
           if( pivot != k ) {
              cout << "pivoting" << endl;
               for( int jj = 0; jj < matrixSize; jj++ ) {
             // swap the rows in the original matrix
                   dummy = semiVario[pivot][jj];
                   semiVario[pivot][jj] = semiVario[k][jj];
                   semiVario[k][jj] = dummy;
            // now swap the rows of the identity matrix
                   dummy = semiVarioInv[pivot][jj];
                   semiVarioInv[pivot][jj] = semiVarioInv[k][jj];
              semiVarioInv[k][jj] = dummy;
               }
           }

        // normalize the row
           dummy = semiVario[k][k];
           for( int j = 0; j < matrixSize; j++ ) {
               if( !dummy ) {
                   cout << "Error - divide by zero" << endl;
                   exit( 1);
             }
               semiVario[k][j] = semiVario[k][j]/dummy;
               semiVarioInv[k][j] = semiVarioInv[k][j]/dummy;
           }
           for( int i = 0; i < matrixSize; i++ ) {
               if( i!=k ) {
                   dummy = semiVario[i][k];
                   for( int j = 0; j < matrixSize; j++ ) {
                       semiVario[i][j] = semiVario[i][j]-
dummy*semiVario[k][j];
                     semiVarioInv[i][j] = semiVarioInv[i][j]-
dummy*semiVarioInv[k][j];
                }
```

```
            }
            }
        }

        // invert the matrix
        for( q = 0; q<matrixSize; q++ ) {
            for(int p = 0; p<matrixSize; p++ ) {
             printf( "%2.6f ", semiVarioInv[p][q] );
            }
            printf( "\n");
        }

        for( int j = 0; j < yrange; j++ ) {
        glat = MIN_LAT+LAT_STEP*j;
            for( int i = 0; i < xrange; i++ ){
                glon = MIN_LON+LON_STEP*i;
                // now load the D matrix
                for( n = 0; n < numOfSensors[z]; n++ ) {
                    distance = dist( obsWind.sensorInfo[sensor-
>index[z][n]].latitude,
                        obsWind.sensorInfo[sensor->index[z][n]].longitude,
glat, glon);
                    d[n] = coVariance(distance);
                }
                d[n] = 1.0;

                // multiply the matrix together
                // to determine the weights
                for( int p = 0; p < matrixSize; p++ ) {
                    for( int q = 0; q < 1; q++ ) {
                        sum = 0.0;
                        for( int k = 0; k < matrixSize; k++ )
                            sum += semiVarioInv[p][k]*d[k];
                  w[p] = sum;
                }
                }

                // initialize the analysed field
                tempu = tempv = 0.0;

              // now step through the sensors to calcualate the estimate
                for( n = 0; n < numOfSensors[z]; n++ ) {
                  obsWind.currObs->data[sensor-
>index[z][n]][z].getWind(&obsu,&obsv);
                  tempu = tempu + w[n]*obsu;
                  tempv = tempv + w[n]*obsv;
                }
                anal1->data[i][j][z].setWind( tempu, tempv );
            }
        }
    }
}

// function: dist
//  This function accepts in the lat lon pairs and calculates the
```

```
// distance between them in degrees squared
float Kriging::dist( float lat1, float lon1, float lat2, float lon2 ) {
//   cout << lat1 << " " << lon1 << " " << lat2 << " " << lon2 << endl;
   return (pow((lat2-
lat1),2.0)+pow(cos((double)(lat1+lat2)*PI/(2.0*180.0)),2.0)*
            pow((lon2-lon1),2.0));
}

#define C1 9.9
#define C0 4.3
#define A 0.44

// function: coVariance
//  This function accepts distance between them in degrees squared
// and calculates the covariance values
float Kriging::coVariance( float dist ) {
//   cout << "co ";
   if( dist )
      return ( C1*(exp( -3.0*dist/A )));
   else
     return( C1 + C0 );
}

// function: displayAnalField
// This function gets the user input for the level and then
// displays the analysed winds at the level specified
void Kriging::displayAnalField() {
   float tempu, tempv;
   int level;

   for( int z = 0; z < NUM_HEIGHTS; z++ )
      cout << numOfSensors[z] << " " ;

   cout << "enter level? ";
   cin >> level;
   while( level != 99 ) {
      for( int j = 0; j < YRANGE; j++ ) {
         for( int i = 0; i < XRANGE; i++ ) {
            anal1->data[i][j][level].getWind( &tempu, &tempv );
            printf( "% 2.2f ",tempu );
            //printf( "% 2.2f/% 2.2f ",tempu,tempv );
        }
         printf( "\n");
   }
      cout << "enter level? ";
      cin >> level;
   }
}
// function: output2dAnalFieldHdr
// This function prints the required information concerning number of
time steps
// size of the output data fields, lat/lon data, lat/lon step size, ...

int Kriging::output2dAnalFieldHdr( char * filename ) {
   FILE * outdatafile;
```

```
    obsListType * obsStepPtr;

    // open the file -- overwrite any existing file

   outdatafile = fopen( filename, "w" );

   // check to make sure the file opened
   if( !outdatafile ) {
      cout << "File could not be opened" << endl;
      exit(1);      // exit loop and program
    }
cout << "Printing Header info to" << filename << endl;
    // now print the appropriate data
   // number of time steps -- 50 max
    fprintf( outdatafile, "%d\n", numOfTimeSteps );
   // number of variables
    fprintf( outdatafile, "2\n");
   // the data array dimmensions
    fprintf( outdatafile, "%d %d %d\n", XRANGE, YRANGE, NUM_HEIGHTS);
    // variable names
    fprintf( outdatafile, "U\nV\n" );
    // time stamps for each time step
    // must loop through and print
    obsStepPtr = obsWind.obsListHead;
    while( obsStepPtr ) {
        fprintf( outdatafile, "%ld ", obsStepPtr->observation->time );
      obsStepPtr = obsStepPtr->next;
    }
    fprintf( outdatafile, "\n" );
    // date stamp for each data set
    // must loop through and print
    obsStepPtr = obsWind.obsListHead;
    while( obsStepPtr ) {
        fprintf( outdatafile,"%ld ", obsStepPtr->observation->date );
      obsStepPtr = obsStepPtr->next;
   }
   // northern most latitude
    fprintf( outdatafile, "\n%f\n", MAX_LAT);
   // the size of the incriment for latitude steps
    fprintf( outdatafile, "%f\n", LAT_STEP );
    // western most longitude
    fprintf( outdatafile, "%f\n",MIN_LON-360.0 );
   // the size of the incriment for latitude steps
    fprintf( outdatafile, "%f\n", LON_STEP);
   //height of the lowest data AGL -- 12 feet
    fprintf( outdatafile, "0.0036576\n");
    fprintf( outdatafile, "%f\n", ALT_STEP*0.0003048);
    fclose( outdatafile );
    return 1;
}

// function: output2dAnalField
// This function prints the actual data to the first pass file.  This
data has not
// been interpolated vertically to the equally spaced vertical grid
```

```
int Kriging::output2dAnalField( char * filename) {
    FILE * outdatafile;
   float outu, outv;

   // begin the outer loop -- allows multiple files to be read in
   // open the file to append to it

   outdatafile = fopen( filename, "a" );

   // check to make sure the file opened
   if( !outdatafile ) {
      cout << "File could not be opened" << endl;
      exit(1);     // exit loop and program
    }
    // first print the u value for every height
  for( int z = 0; z < NUM_HEIGHTS; z++ ) {
       for( int j = 0; j < YRANGE; j++ ) {
           for( int i = 0; i < XRANGE; i++ ) {
              anal1->data[i][j][z].getWind( &outu, &outv );
           fprintf( outdatafile, "% f ", outu );
           }
         fprintf( outdatafile, "\n" );
       }
   }
   // then print out the v value at each height
    for( z = 0; z < NUM_HEIGHTS; z++ ) {
       for( int j = 0; j < YRANGE; j++ ) {
           for( int i = 0; i < XRANGE; i++ ) {
              anal1->data[i][j][z].getWind( &outu, &outv );
           fprintf( outdatafile, "% f ",outv );
           }
         fprintf( outdatafile, "\n" );
       }
    }
    fclose( outdatafile );
    return( 1 );
}
float Kriging::semiVarioFunc( float dist ) {
   return dist;
}

//  start the main function -- this is the driver for the
//     entire program
int main( void ) {
   int datain;

    cout << "Display Data? (1=yes, 0=no): ";
    cin >> display;
    cout << "Collect Data? (1=yes, 0=no): ";
    cin >> dataCollection;
    cout << "Method? (1=Barnes, 2=Kriging): ";
    cin >> datain;

    if( datain == 1 )
```

```
        callBarnes();
    else if( datain == 2 )
        callKriging();
    else
      cout << "Invalid selection.";
    cout << "I can do all things through Christ who strengthens me!" ;
   return 1;
}
void callKriging( void ) {
    Kriging anal2Wind;
    cout << "Kriging Test1" << endl;
    anal2Wind.doKriging();
    cout << "After do Kriging" << endl;
    anal2Wind.displayAnalField();
    anal2Wind.~Kriging();

    doVerticalInterpolation();

}

void callBarnes( void ) {
   Barnes analWind;
    cout << "Barnes Test1" << endl;
    analWind.doBarnes();
    analWind.displayAnalField();
    analWind.~Barnes();
    doVerticalInterpolation();

}
```

Appendix B: Wind Tower Data

ID	Lat	Lon	Height
061	28 30 46.8000	279 26 19.3200	2.52
1102	28 34 10.9200	279 25 48.9600	2.43
3132	28 37 32.1600	279 21 34.4400	0.00
0001	28 26 1.8012	279 25 35.7837	3.21
0003	28 27 35.3242	279 28 23.8319	3.66
0019	28 44 36.5204	279 17 58.0930	0.60
0022	28 47 50.9066	279 15 43.9209	0.17
0036	28 28 19.5471	279 27 38.4626	2.08
0040	28 33 43.8099	279 25 17.2069	3.03
0041	28 35 1.100	279 24 56.7270	2.90
0108	28 32 9.0653	279 25 30.7496	2.91
0112	28 36 50.7623	279 22 46.7753	2.01
0303	28 27 35.925	279 25 43.95	2.5
0311	28 36 9.8910	279 21 31.04	2.3
0393	28 36 37.7536	279 23 35.2661	3.44
0394	28 36 20.3539	279 23 54.1554	3.46
0397	28 37 45.8793	279 22 35.285	2.36
0398	28 37 29.3062	279 22 54.5717	2.04
0403	28 27 30.86	279 24 27.672	2.60
0412	28 36 22.5160	279 19 34.0590	2.9
0415	28 39 30.8617	279 18 0.6606	2.43
0418	28 42 19.9533	279 16 24.7431	0.88
0421	28 46 31.6775	279 11 44.4169	1.40
0506	28 30 56.9663	279 21 36.0032	1.73
0509	28 33 44.2202	279 19 50.1415	2.24
0511	28 35 54.8076	279 19 6.0396	0.79
0512	28 36 57.6467	279 18 25.0582	1.28
0513	28 37 50.7106	279 17 50.2348	3.16
0714	28 38 35.2840	279 15 6.5332	2.35
0803	28 27 47.4360	279 19 47.3590	3.30
0819	28 44 47.0571	279 7 45.4213	8.14
1000	28 24 28.5062	279 14 22.6214	9.14
1007	28 31 37.8450	279 13 32.8740	1.1
1012	28 36 20.2247	279 10 30.7188	9.17
1204	28 29 3.4308	279 12 51.9385	8.11
1500	28 24 41.2004	279 4 17.6290	8.07
1612	28 37 2.3245	279 2 30.9478	3.14
1617	28 40 34.3274	279 0 4.5389	1.85
2008	28 31 23.2770	278 59 24.1613	15.6
2016	28 38 56.0577	278 55 50.4109	7.39
2202	28 26 30.0407	278 58 15.1926	20.09
9001	28 23 35.5266	279 10 43.8410	6.65
9404	28 20 17.4517	279 16 4.4544	6.43
1605	28 29 45.4558	279 6 56.6325	2.98
1108	28 32 29.688	279 11 18.24	0.00
0110	28 34 10.989	279 24 48.909	2.5
0805	28 31 6.0920	279 18 13.5320	2.5
062	28 30 46.8000	279 26 19.3200	2.52

```
1101 28 34 10.9200 279 25 48.9600 2.43
3131 28 37 32.1600 279 21 34.4400 0.00
```

Appendix C: Distance Calculations

The distances computed in this thesis are computed in latitude degrees. They are based upon the approximations that Δxwy and Δxyz in Figure 19 are right triangles. This allows the distance to be written:

$$D^2 = a^2(\Delta\phi)^2 + \left(a\cos\left(\bar{\phi}\right)\right)^2(\Delta\lambda)^2,$$

where a is the earth's radius, $\Delta\phi = \phi_2 - \phi_1$, $\bar{\phi} = \frac{1}{2}(\phi_1 + \phi_2)$, and $\Delta\lambda = \lambda_2 - \lambda_1$. This equation can be rearranged to the following form:

$$d = \frac{D^2}{a^2} = (\Delta\phi)^2 + \cos^2\bar{\phi}(\Delta\lambda)^2.$$

The distances remained squared because Barnes' method used them squared. They were not changed for Kriging to ensure any inaccuracy resulting from the approximation was applied equally to both methods.

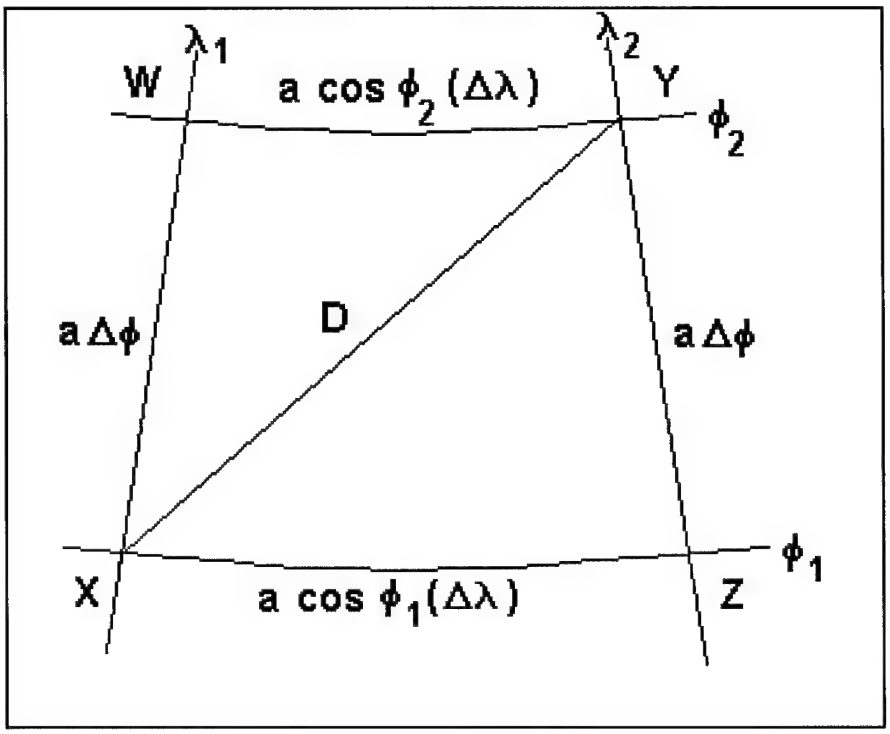

Figure 19 – Distance Approximation Figure
This figure illustrates the approximations used in the calculation of the distances on a spherical surface. The distance, D, is being calculated using Pythagorean's Theorem

Bibliography

Air Force Manual 15-125, 11 August 1997: *Weather Station Operations.* Department of the Air Force, Washington DC,11.

Barnes, S.L., 1964: A technique for maximizing details in numerical weather map analysis. *J. Applied Met.,* 3, 396-409.

Chapra, S. C., and Raymond P. Canale, 1988: *Numerical Methods for Engineers, Second Edition.* McGraw-Hill Publishing Co., 839 pp.

Cheney, W., and D. Kincaid, 1985: *Numerical Mathematics and Computing, Second Edition.* Brooks/Cole Publishing Co., 430 pp.

Cressman, G.P., 1959: An operative objective analysis scheme. *Mon. Wea. Rev.,* 87, 367-374.

Clark, I., 1987: *Practical Geostatistics.* Elsevier Applied Science Publishers LTD, 129 pp.

Daley, R., 1991: *Atmospheric Data Analysis.* Cambridge University Press, 457 pp.

Garret, J. R., 1992: *The atmospheric boundary layer.* Cambridge University Press, 316 pp.

Isaaks, E. H., and R. Mohan Srivastava, 1989: *An Introduction to Applied Geostatistics.* Oxford University Press, 561 pp.

Kreyszig, Erwin, 1993: *Advanced Engineering Mathematics.* John Wiley & Sons, Inc., 1271 pp.

Mendenhall, W. and Terry Sincich, 1992: *Statistics for Engineering and the Sciences, Third Edition.* Dellen Publishing Co., 963 pp.

Mullen, Steven L. Class Handout, Atmo 471, Synoptic Meteorology, University of Arizona, Tucson AZ, Fall Semester 1993.

Roeder, William P. "Operational Research Requirements For America's Space Launch Program at 45th Weather Squadron." Addresses to Air Force Institute of Technology Meteorology students and faculty. Air Force Institute of Technology, Wright-Patterson AFB OH 5 Feb 1998

Sen, Zakai, 1997: Objective analysis by cumulative semivariogram technique and its application in tTurkey. *Journal of Applied Meteorology,* 36, 1712-1724.

Williams, Robert. Air Force Weather Agency, Offutt AFB NE. Personal Correspondence, 16 September 1998.

Vita

First Lieutenant Michael W. Engel was born on 12 September 1965 at the United States Air Force Academy, Colorado. He graduated from Sahuaro High School, Tucson, Arizona, in 1983. He entered active duty on July of 1984, attending the Law Enforcement Specialist course as well as the Air Base Ground Defense course before being assigned to the 3380 Security Police Squadron, Keesler AFB, MS in January 1985. In 1991, he was selected for the Airman Education Commissioning Program. He graduated from the University of Arizona, Tucson, Arizona, in 1995 with a Bachelor of Science in Atmospheric Science. Upon graduation, 5 May 1995, he was commissioned through Officer Training School as a Distinguished Graduate. In May of 1995 he began the Interim Officers Course, Keesler Air Force Base, Mississippi. In July 1995, he was assigned to the 27 Operations Support Squadron, Cannon AFB, NM, as the Wing Weather Officer. In January of 1997 he became the Officer in Charge of Weather Station Operations.

In September 1997, Lieutenant Engel entered the School of Engineering, Air Force Institute of Technology, Wright Patterson AFB OH.

Lieutenant Engel is married to the former Kristi M. Crouch of Tucson, Arizona and has four children, Rebecca (13), Kevin (11), Lesli (8), and Jami (6).

Permanent Address: 15 N Gollob Rd
Tucson, AZ, 85710

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	3 March 1999	Final

4. TITLE AND SUBTITLE
Evaluation of Barnes' Method and Kriging for Estimating the Low Level Wind Field

5. FUNDING NUMBERS

6. AUTHOR(S)
1st Lt Michael W. Engel

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Lt Col Cecilia A. Miner
AFIT/ENP
2950 P Street
Wright-Patterson AFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GM/ENP/99M-05

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
William P. Roeder
45WS/SYR
1201 Minuteman Street
Patrick AFB, FL 32925-3238

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis evaluates two different methods of estimating a three dimensional wind field based upon a limited number of irregularly-spaced observations. This work was performed for the 45th Weather Squadron to determine how well the two methods worked and their potential for use in a visualization program. The two methods evaluated were Barnes' method and a method called Kriging, which is commonly used in geostatistics. Both of these estimation techniques were implemented and then evaluated to determine how accurate the estimates were that they created. The methods' accuracies were determined by withholding an observation from the observed wind field data set, performing the estimation, and then comparing the estimated value at the point of the withheld observation with the actual value withheld. These performance results were compared to determine which method produced a more accurate estimated wind field. Barnes' method proved to be the less complicated to implement, but Kriging provided a more accurate estimate. Both of the methods had a significant amount of estimation error associated with them. This large error casts serious doubt on their abilities to produce an accurate enough estimation to be useful in analyzing the low-level wind field.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Objective Analysis, Kriging, Barnes, Interpolation	122
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL